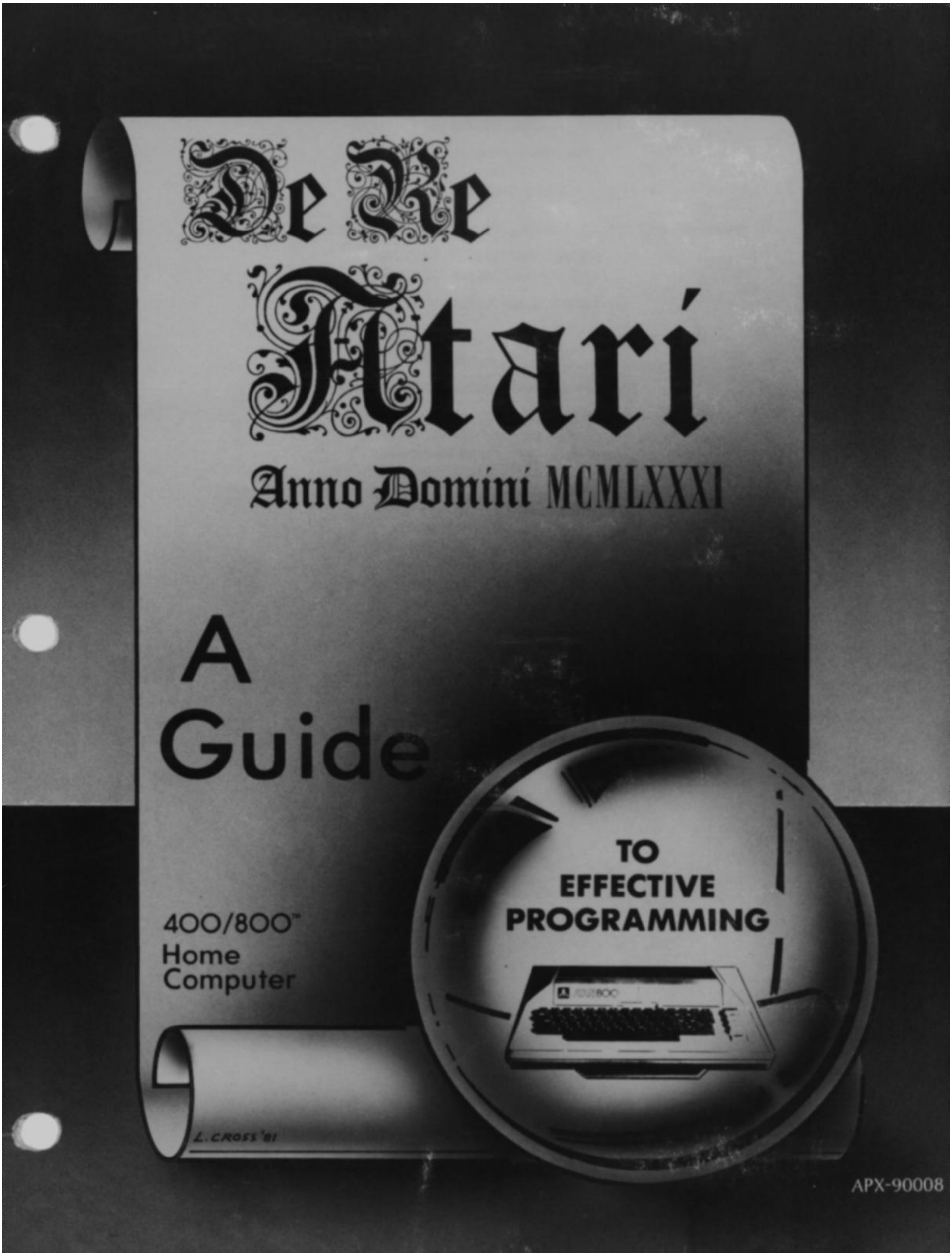


De Re Atari

Converted by Andreas Bertelmann for ABBUC - www.abbuc.de

Preface

Introduction

by enter value here

This manual is about the ATARI Home Computer. It covers both the ATARI 400TM and the ATARI 800TM Computers. These two computers are electrically identical, differing only in mechanical features such as the keyboards and cartridge slots. The purpose of this manual is to explain in detail how to use all the features of the ATARI Computer. Because this is a complex and powerful machine, the explanations are accordingly rather long. Furthermore, they demand some expert Expertise on the part of the reader. This book is not intended for the beginning programmer. The reader should be thoroughly familiar with the BASIC Reference Manual, which is provided with the computer. Familiarity with assembly language is also essential. A glossary in the back defines and explains some of the less commonly encountered jargon. However, this glossary does not include terms that every serious personal computer programmer should already know.

Written as a training manual for professional programmers who use the ATARI Home Computer, this book may be modified for general use at some later date. It does not supplant the technical reference manual (ATARI part number C016555), which is a reference for programmers who already understand the system. This book is intended to be a tutorial that explains ideas and possibilities rather than defining registers and control codes.

The title, DE RE ATARI, is pronounced "Day Ray Atari". It is an obscure literary reference. Some Latin manuscripts in Roman and medieval times were entitled "De Re This" or "De Re That". Thus, "De Re Rustica" was a poem on farming and "De Re Metallica" described metallurgy. Loosely translated, "De Re" means "All About".

Most of the word processing for the book was carried out with Atari computers. A source file editor was used for text editing, and a modified version of FORMS (available from the Atari Program Exchange) was used to format and print the text. A letter-quality printer was used for output. Some sections were developed with a conventional word processor.

The Software Development Support Group wrote this book. Chris Crawford wrote Sections 1 through 6 and Appendices A and B. Lane Winner wrote Section 10 and Appendix D with assistance from Jim Cox. Amy Chen wrote Appendix C. Jim Dunion wrote Sections 8 and 9. Kathleen Pitta wrote Appendix E. Bob Fraser wrote Section 7. Gus Makreas prepared the Glossary. The final result has many flaws, but we are proud of it.

Content

Chapter I

System Overview

1 System Overview

The ATARI Home Computer is a second-generation personal computer. First and foremost, it is a consumer computer. The thrust of the design is to make the consumer comfortable with the computer. This consumer orientation manifests itself in many ways. First, the machine is proofed against consumer mistakes stakes by such things as polarized connectors that will not go in the wrong way, a power interlock on the lid to the internal electronics, and a pair of plastic shields protecting the SYSTEM RESET key. Second, the machine has a great deal of graphics power; people respond to pictures much more readily than to text. Third, the machine has strong sound capabilities. Again, people respond to direct sensory input better than to indirect textual messages. Finally, the computer has joysticks and paddles for more direct tactile input than is possible with keyboards. The point here is not that the computer has lots of features but rather that the features are all part of a consistent design philosophy aimed directly at the consumer. The designer who does not appreciate this fundamental fact will find himself working against the grain of the system.

The internal layout of the ATARI 400/800Tm Computer is very different from other systems. It of course has a microprocessor (a 6502), RAM, ROM, and a (PIA). However, it also has three special- purpose (LSI) chips known as ANTIC, CTIA, and POKEY. These chips were designed by Atari engineers primarily to take much of the burden of housekeeping off of the 6502, thereby freeing the 6502 to concentrate on computations. While they were at it, they designed a great deal of power into these chips. Each of these chips is almost as big (in terms of silicon area) as a 6502, so the three of them together provide a tremendous amount of power. Mastering the ATARI 400/800 Computers is primarily a matter of mastering these three chips.

ANTIC is a microprocessor dedicated to the television display. It is a true microprocessor; it has an instruction set, a program (called the display list), and data. The display list and the display data are written into RAM by the 6502. ANTIC retrieves this information from RAM using direct memory access (DMA). It processes the higher level instructions in the display list and translates these instructions into a real-time stream of simple instructions to CTIA.

CTIA is a television interface chip. ANTIC directly controls most of CTIA's operations, but the 6502 can be programmed to intercede and control some or all of CTIA's functions. CTIA converts the digital commands from ANTIC (or the 6502) into the signal that goes to the television. CTIA also adds some factors of its own, such as colour values, player-missile graphics, and col- lision detection.

POKEY is a digital input/output (I/O) chip. It handles such disparate tasks as the serial I/O bus, audio generation, keyboard scan, and random number generation. It also digitizes the resistive paddle inputs and controls maskable interrupt (IRQ) requests from peripherals.

All four of these LSI chips function simultaneously. Careful separation of their functions in the design phase has minimized conflicts between the chips. The only hardware level conflict between any two chips in the system occurs when ANTIC needs to use the address and data buses to fetch its display information. To do this, it halts the 6502 and takes control of the buses.

As with all 6502 systems, the I/O is memory-mapped. Figure 1-1 presents the coarse memory map for the computer. Figure 1-2 shows the hardware arrangement.

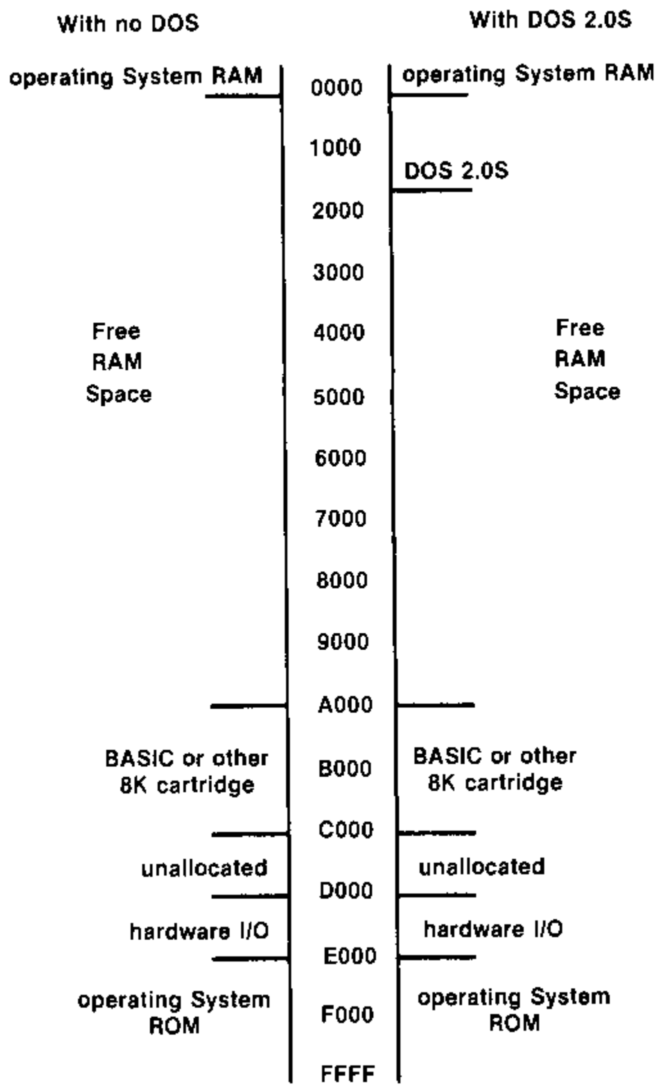
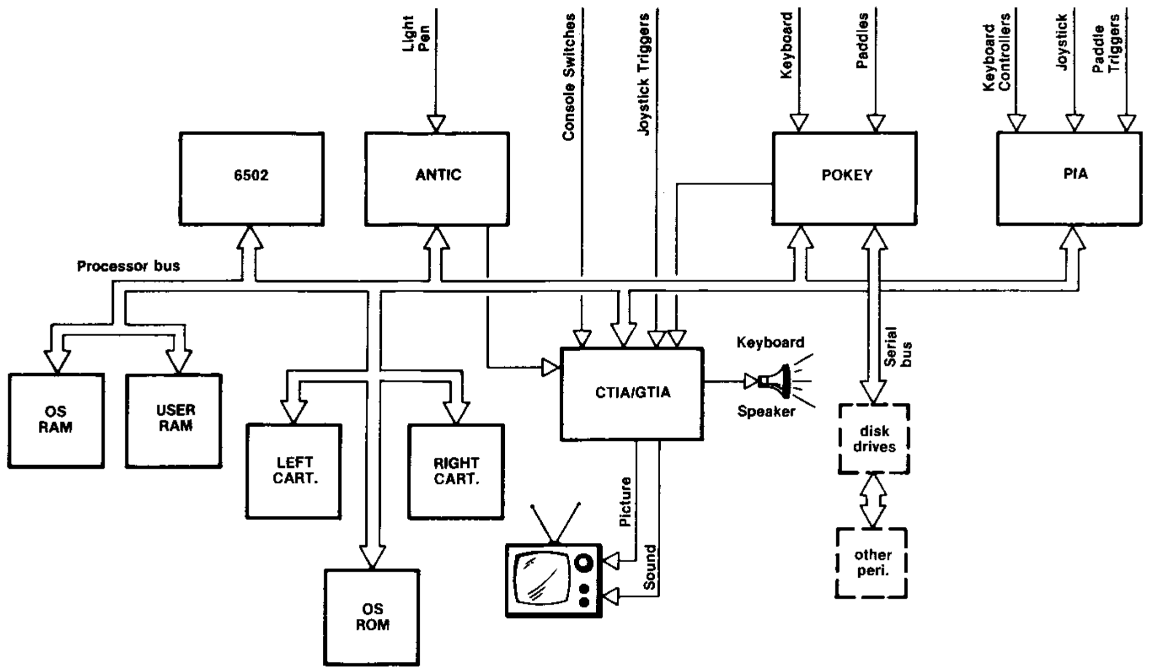


Figure 1-1 Atari Memory Arrangement



ATARI 400/800

Figure 1-2 Atari Hardware Arrangement

Chapter II

Antic and the display list

2 Antic and the display list

TELEVISION DISPLAYS
To understand the graphics capabilities of the ATARI Home Computer, one must first understand the rudiments of how a television set works. Television sets use what is called a raster scan display system. An electron beam is generated at the rear of the television tube and shot toward the television screen. Along the way, it passes between sets of horizontal and vertical coils which, if energized, can deflect the beam. In this way the beam can be made to strike any point on the screen. The electronics inside the television set cause the beam to sweep across the television screen in a regular fashion. The beam's intensity can also be controlled. If you make the beam more intense, the spot in the screen that it strikes will glow brightly; if you make it less intense, the spot will glow dimly or not at all

The beam starts at the top-left corner of the screen and traces horizontally across the screen. As it sweeps across the screen, its changes in intensity paint an image on the screen. When it reaches the right edge of the screen, it is turned off and brought back to the left side of the screen. At the same time it is moved down just a notch. It then turns back on and sweeps across the screen agate. This process is repeated for a total of 262 sweeps across the screen. (There are actually 525 sweeps across the screen in an alternating system known as "interlace." We will ignore interlace and act as if the television has only 262 lines.) These 262 lines fill the screen from top to bottom. At the bottom of the screen (after the 262nd line is drawn), the electron beam is turned off and returned to the upper left corner of the screen. Then it starts the cycle all over agate. This entire cycle happens 60 times every second.

Now for some Jargon: a single trace of the beam across the screen is called a "horizontal scan line." A horizontal scan line is the fundamental unit of measurement of vertical distance on the screen. You state the height of an image by specifying the number of horizontal scan lines it spans. The period during which the beam returns from the right edge to the left edge is called the "horizontal blank." The period during which the beam returns to the top of the screen is called the "vertical blank." The entire process of drawing a screen takes 16,684 microseconds. The vertical blank period is about 1400 microseconds. The horizontal blank takes 14 microseconds. A single horizontal line takes 64 microseconds.

Most television sets are designed with "overscan"; that means they spread the image out so the picture edges are off the edge of the television tube. This guarantees that you have no unsightly borders in your television picture. It is very bad for computers, though, because screen information that is off the edge of the picture does you no good. For tints reason the picture that the computer puts out must be somewhat smaller than the television can theoretically display. Therefore, only 192 horizontal scan lines are normally used by the ATARI display. Thus, the normal limit of resolution of a television set used with this computer is 192 pixels vertically.

The standard unit of horizontal distance is the "color clock." You specify the width of an image by stating how many color clocks wide it Is. There are 228 color clocks in a single horizontal scan line, of which a maximum of 176 are actually visible. Thus, the ultimate limit for full-color horizontal resolution with a standard color television is 176 pixels. It is possible with the ATARI Home Computer System to go even finer and control individual half- clocks. This gives a horizontal resolution ton of 352 pixels. However, use of tints feature will produce interesting color effects known as color artifacts. Color artifacts can be a nuisance if they are not desired; they can be a boon to the programmer who desires additional color and is not fazed by their restrictions.

COMPUTERS AND TELEVISIONS
The fundamental problem any microcomputer has In using a raster scan television for display purposes is that the television display is a dynamic process; because of this, the television does not remember the image. Consequently, the computer must remember the screen Image and constantly send a signal to the television telling it what to display. This process of sending in format ton to the television is a continuous process and it requires full-time attention For tints reason most microcomputers have special hardware circuits that handle the television The basic arrangement is the same on virtually all systems:

microprocessor-->screen RAM--->video hardware--->TV screen

The microprocessor writes information to the screen RAM area that holds the screen data. The video hardware is constantly dipping into this RAM area, gettting screen data that it converts into television signals These signals go to the television which then displays the information. The screen memory is mapped onto the screen in the same order that it follows in RAM. That is, the first byte in the screen memory maps to the top-left corner of the screen, the second byte maps one position to the right, then the third, the fourth, and so on to the last byte which is mapped to the lower right corner of the screen.

The quality of the Image that gets to the screen depends on two factors: the quality of the video hardware, and the quantity of screen RAM used for the display The simplest arrangement is that used by TRS- 80 and PET. (TRS-80) is a trademark of Radio Shack Co; PET is a trademark of Commodore Business Machines.) Both of these machines allocate a specific 1K of RAM as screen memory. The video hardware circuits imply pull data out of tints area, Interpret it as characters (using a character set in ROM), and put the resulting characters onto the screen. Each byte represents one character, allowing a choice of 256 different characters in the character set. With 1K of screen RAM, one thousand characters can be displayed on the screen. There isn't much that can be done with tints arrangement. The Apple uses more advanced video hardware. (Apple is a trademark of Apple Computers.) Three graphics modes are provided: text, lo-resolutton graphics, and hi- resolution graphics. The text graphics mode operates in much the same way that the PET and TRS-80 displays operate. In the low-resolution graphics mode, the video hardware reaches into screen memory and interprets it differently. Instead of interpreting each byte as a character, each byte is interpreted as a pair of color nibbles. The value of each nibble specifies the color of a single pixel. In the high-resolution graphics mode each bit in screen memory is mapped to a single pixel. If the bit is on, the pixel gets color in It; if the bit is off, the pixel stays dark. The situation is complicated by a variety of design nuances in the Apple, but that is the basic idea. The important idea is that the Apple has three display modes; three completely different ways of interpreting the data in screen memory. The Apple video hardware is smart enough to interpret a screen memory byte as either an 8-bit character (text mode), two 4-bit color nybbles (lo-resolution mode), or 7 individual bits for a bit map hi-resolution mode).

ANTIC, A VIDEO MICROPROCESSOR

The ATARI 400/800 display list system represents a generalization of these systems. Where PET and TRS-80 have one mode and Apple has three modes, the ATARI 400/800 has 14 modes. The second important difference is that display modes can be mixed on the screen. That is, the user is not restricted to a choice between a screen ful of text or a screenful of graphics. Any collection of the 14 graphics. modes can be displayed on the screen. The third important difference is that the screen RAM can be located anywhere in the address space of the computer and moved around while the program is running, while the other machines use fixed-screen RAM areas.

All of this generality is made possible by a video microprocessor called ANTIC. Where the earlier systems used rather simple video circuitry, Atari designed a full-scale microprocessor just to handle the intricacies of the television display. ANTIC is a true microprocessor; it has an instruction set, a program, and data. The program for ANTIC is called the display list. The display list specifies three things: where the screen data may be found, what display modes to use to Interpret the screen data, and what special display options (if any) should be implemented. When (using the display list, it is important to shed the old view of a screen as a homogeneous image in a single mode and see it instead as a stack of "mode lines." A mode line is a collection of horizontal scan lines. It stretches horizontally all the way across the screen. A Graphics 2 mode line is 16 horizontal scan lines high, while a Graphics 7 mode line is only two scan lines high. Many graphics. modes available from BASIC are homogeneous; an entire screen of a single mode is set up. Do not limit your imagination to this pattern; with the display list you can create any sequence of mode lines down the screen. The display list is a collection of code bytes that specify that sequence.

ANTIC'S instruction set is rather simple. There are four classes of instructions: map mode, character mode, blank line and jump. Map mode instructions cause ANTIC to display a mode line with simple colored pixels (no characters). Character mode instructions cause ANTIC to display a mode line with characters in it. Blank line instructions cause ANTIC to display a number of horizontal scan lines with solid background color. Jump instructions are analogous to a 6502 JMP instruction; they reload ANTIC's program counter.

There are also four special options that can sometimes be specified by setting a designated bit in

the ANTIC instruction. These options are: display list Interrupt (DLI), load memory scan (LMS), vertical scroll, and horizontal scroll.

Map mode instructions cause ANTIC to display a mode line containing pixels with solid color in them. The color displayed comes from a color register. The choice of color register is specified by the value of the screen data. In four-color map modes (BASIC modes 3, 5, and 7, and ANTIC modes 8, A, D, and E), a pair of bits is required to specify a color:

Value of Bit Pair	Color Register Used
00 0	COLBAK
01 1	COLPFO
10 2	COLPF1
11 3	COLPF2

Since only two bits are needed to specify one pixel, 4 pixels are encoded in each screen data byte. For example, a byte of screen data containing the value $1B would display 4 pixels; the first would be the background, the second would be color register 0, the third would be color register 1, and the fourth would be color register 2:

$1B = 00011011 = 00 01 10 11

In two-color map modes (BASIC modes 4, 6, and 8, and ANTIC modes 9, B. C, and F) each bit specifies one of two color registers. A bit value of 0 selects background color for the pixel and a bit value of 1 selects color register 0 for the pixel. Eight pixels can be stored in one screen data byte.

There are eight different map display modes. They differ in the number of colors they display (2 vs 4), the vertical size one mode line occupies (1 scan line 2, 4, or 8), and the number of pixels that fit horizontally into one mode line (40, 80, 160, or 320). Thus, some map modes give better resolution; these will of course require more screen RAM. Figure 2-1 presents this information for all modes.

ANTIC Mode	BASIC Mode	No. Colors	Scan Lines Mode Line	Pixels Mode Line	Bytes Line	Bytes/ Screen
2	0	2	8	40	40	960
3	none	2	10	40	40	760
4	none	4	8	40	40	960
5	none	4	16	40	40	480
6	1	5	8	20	20	480
7	2	5	16	20	20	240
8	3	4	8	40	10	240
9	4	2	4	80	10	480
A	5	4	4	80	20	960
B	6	2	2	160	20	1920
C	none	2	1	160	20	3840
D	7	4	2	160	40	3840
E	none	4	1	160	40	7680
F	8	2	1	320	40	7680

Figure 2-1 ANTIC Mode Line Requirements

Character mode instructions cause ANTIC to display a mode line with characters in it. Each byte in screen RAM specifies one character. There are six character display modes. Character displays are discussed in Section

Blank line instructions produce blank lines with solid background color. There are eight blank line instructions they specify skipping one through eight blank lines

There are two jump instructions. The first (JMP) is a direct jump; it. reloads ANTIC's program counter with a new address that follows the JMP instruction as an operand. Its only function is to provide a solution to a tricky problem: ANTIC's program counter has only 10 bits of counter and six bits of latch and so the display list cannot cross a 1K boundary. If the display list must cross a 1K boundary then it must use a JMP instruction to hop over the boundary. Note that this means that display lists are not fully relocatable.

The second jump instruction (JVB) is more commonly used. It reloads the program counter with the value in the operand and waits for the television to perform a vertical blank. This instruction is normally used to terminate a display list by jumping back up to the top of the display list Jumping up to the top of the display list turns it into an infinite loop; waiting for vertical blank ensures that the infinite loop is synchronized to the display cycle of the television. Both JMP and JVB are 3-byte instructions the first byte is the opcode, the second and third bytes are the address to jump to (low then high).

The four special options mentioned previously will be discussed in Sections 5 and 6. The load memory scan (LMS) option must have a preliminary explanation. This option is selected by setting bit 6 of a map mode or a character mode instruction byte. When ANTIC encounters such an instruction, it will load its memory scan counter with the following 2 bytes. This memory scan counter tells ANTIC where the screen RAM is. It will begin fetching display data from this area. The LMS instruction is a 3- byte instruction: 1 byte opcode followed by 2 bytes of operand. In simple display lists the LMS instruction Is used only once, at the beginning of the display list. It may sometimes be necessary to use a second LMS instruction. The need arises when the screen RAM area crosses a 4K boundary. The memory scan counter has only 12 bits of counter and 4 bits of latch; thus, the display data cannot cross a 4K boundary. In this case an LMS instruction must be used to jump the memory scan counter over the boundary. Note that this means that display data is not fully relocatable. LMS instructions have wider uses which will be discussed later.

BUILDING DISPLAY LISTS
Every display list should start off with three "blank 8 lines" instructions. This is to defeat vertical overscan by bringing the beginning of the display 24 scan lines down. After this is done, the first display line should be specified. Simultaneously, the LMS should be used to tell ANTIC where it will find the screen RAM. Then follows the display list proper, which lists the display bytes for the mode lines on the screen. The total number of horizontal scan lines produced by the display list should always be 192 or less; ANTIC does not maintain the screen timing requirements of the television. If you give ANTIC too many scan lines to display it will do so, but the television screen will probably roll. Displaying fewer than 192 scan lines will cause no problems; indeed, it will decrease 6502 execution time by reducing the number of cycles stolen by ANTIC. The programmer must calculate the sum of the horizontal scan lines produced by the display list and verify it. The display list terminates with a JVB instruction. Here is a typical display list for a standard BASIC Graphics mode 0 display (all values are in hexadecimal):

```
70 Blank 8 lines
70 Blank 8 lines
70 Blank 8 lines
42 display ANTIC mode 2 (BASIC mode0)
20 Also, screen memory starts at7C20
7C
02 Display Antic Mode 2
02
02
02
02
02
02
02
02
02
02
02
02
02
02
02
02
02
02
02
```

02
02
02
41 Jump and watt for vertical
E0 blank to display list which
7B starts at $7BEO

As you can see, This display list is short---only 32 bytes. Most display lists are less than 100 bytes long. Furthermore, they are quite simple in structure and easy to set up.

To implement your own display list you must first design the display format. This is best done on paper. Lay out the screen image and translate it. into a sequence of mode lines Keep track of the scan line count of your display by looking up the scan line Requirements of the various modes in Figure 2-1. Translate the sequence of mode lines into a sequence of ANTIC mode bytes. Put three "blank 8 lines bytes ($70) at the top of the list Set bit 6 of the first display byte (that is make the upper nybble a 4). This makes a load memory scan command. Follow with 2 bytes which specify the address of the screen RAM (low then high). Then follow with the rest of the display bytes. At the end of your display list put in the JVB instruction ($41) and the address of the top of the display list Now store all of these bytes into RAM. They can be anywhere you want; just make sure they don't overlay something else and your JVB points to the top of the display list The display list must not cross a 1K address boundary. If you absolutely must have it. cross such a boundary, insert a JMP instruction just in front of the boundary. The JMP instructions's operand is the address of the first byte on the other side of the boundary. Next you must turn off ANTIC for a fraction of a second while you rewrite its display list pointer. Do this by writing a 0 into SDMCTL at location $22F. Then store the address of the new display list into $230 and $231 (low then high). Lastly, turn ANTIC back on with a $22 into SDMCTL. During the vertical blank, while ANTIC is quiet, the operating system (OS) will reload ANTIC's program counter with these values.

WRITING TO A CUSTOM DISPLAY LIST SCREEN
Screen memory can be placed anywhere in the address space of the computer. Normally the display list specifies the beginning of the screen memory with the first display instruction---the initial LMS instruction However, ANTIC can execute a new LMS instruction with each display line of the display list if this is desired. In This way information from all over the address space of the computer can be displayed on a single screen. This can be of value in setting up independent text windows.

There are several restrictions in your placement of the screen memory. First screen memory cannot cross a 4K address boundary. If you cannot avoid crossing a 4K boundary (as would be the case in BASIC mode 8, which uses 8K of RAM) you must reload the memory scan counter with a new LMS instruction. Second, if you wish to use any of the operating system screen routines you must abide by the conventions the OS uses. This can be particularly difficult when using a modified display list in a BASIC program. If you alter a standard display list from a BASIC program and then attempt to PRINT or PLOT to the screen, the OS will do so under the assumption that the display list is unchanged. This will probably result in a garbled display

There are three ways the display can fall when you attempt this. First BASIC may refuse to carry out a screen operation because it is impossible to do in the graphics. mode that the OS thinks it is in. The OS stores the value of the graphics mode that it thinks is on the screen in address $57. You can fool the OS into cooperating by poking a different value there. Poke the BASIC mode number, not the ANTIC mode number.

The second failure you might get arises when you mix mode lines with different screen memory byte Requirements. Some mode lines require 40 bytes per line some require 20 bytes per line and some require only 10 bytes per line. Let's say that you insert one 20-byte mode line into a display list with 40 byte mode lines. Then you PRINT text to the display. Everything above the interloper line is fine, but below it. the characters are shifted 20 spaces to the right. This is because the OS assumed that each line would require 40 bytes and positioned the characters accordingly. But ANTIC, when it encountered the interloper line took only 20 bytes of what the OS thought should be a 40-byte line ANTIC interpreted the other 20 bytes as belonging to the next line and displayed them there. This resulted in the next line and all later lines being shifted 20 spaces to the right.

The only absolute way around This problem is to refrain from (using BASIC PRINTs and PLOTs to output to a custom display list screen. The quick-and-dirty solution is to organize the screen into line groups that contain integer multiples of the standard byte requirement. That is, do not insert a 20-byte mode line into a 40- byte display instead insert two 20-byte lines or one 20-byte line and two 10-byte lines So long as you retain the proper integer multiples, the horizontal shift will be avoided.

This solution accentuates the third problem with indexed display lists and BASIC: vertical shifts. The OS positions screen material vertically by calculating the number of bytes to skip down from the top of the screen. In a standard 40-byte line display, BASIC would position the characters onto the tenth line by skipping 360 bytes from the beginning. If you have inserted four 10-byte lines BASIC will end up three lines further down the screen than you would otherwise expect. Furthermore, different mode lines consume different numbers of scan lines, so the position on the screen will not be quite what you expected if you do not take scan line costs into account.

As you can see, mixed mode displays can be difficult to use in conjunction with the OS. Often you must fool the OS to make such displays work. To PRINT or PLOT to a mode window, POKE the BASIC mode number of that window to address $57, then POKE the address of the top left pixel of the mode window into locations $58 and $59 (low then high). In character modes, execute a POSITION 0,0 to home the cursor to the top-left corner of the mode window. In map modes, all PLOTs and DRAWTOs will be made using the top-left corner of the mode window as the origin of the coordinate system.

The display list system can be used to produce appealing screen displays. Its most obvious use is for mixing text and graphics. For example, you could prepare a screen with a bold BASIC mode 2 title, a medium size BASIC mode 1 subtitle, and small BASIC mode 0 fine print. You could then throw in a BASIC mode 8 picture in the middle with some more text at the bottom. A good example of this technique is provided by the display in the ATARI States and Capitals program.

The aforementioned problems will discourage the extensive use of such techniques from BASIC. with assembly language routines, modified display lists are best used by organizing the screen into a series of windows, each window having its own LMS instruction and its own independent RAM area.

APPLICATIONS OF DISPLAY LISTS
One simple application of display list modifications is to vertically space lines on the screen by inserting blank line bytes. This will add some vertical spacing which will highlight critical messages and enhance the readability of some displays.

Another important use of display list manipulations is in providing access to features not available from BASIC. There are three text modes supported by ANTIC that BASIC does not support. Only display list manipulations gain the user access to these modes. There are also display list interrupt and fine scrolling capabilities that are only available after the display list is modified. These features are the subjects of Sections 5 and 6.

Manipulations with the LMS instruction and its operand offer many possibilities to the creative programmer. For example, by changing the LMS during vertical blank, the programmer can alternate screen images. This can be done at slow speed to change between predrawn displays without having to redraw each one. Each display would continue to reside in (and consume) RAM even while it. is not in use, but it. would be available almost instantly. This technique can also be used for animation. By flipping through a sequence of displays cyclic animation can be achieved. The program to do this would manipulate only 2 address bytes to display many thousands of bytes of RAM.

It is also possible to superimpose images by flipping screens at high speed. The human eye has a time resolution; of about 1/16 of a second, so a program can cycle between four images, one every 1/60 of a second, so that each repeats every 1/15 of a second. In This way, up to four images can appear to reside simultaneously on the screen. Of course, there are some drawbacks to This method. first four separate displays may well cost a lot of RAM. Second, each display image will be washed out because it. only shows up one quarter of the time. This means that the background of all displays must be black, and each image must be bright. Furthermore, there will

be some unpleasant screen flicker when this technique is used. A conservative programmer might consider cycling between only three or even only two images. This technique can also be used to extend the color and luminosity resolution; of the computer. By cycling between four versions of the same image each version stressing one color or luminosity range, a wider range of colors and luminosities is available. For example, suppose you wish to display a bar of many different luminances. First set your four color registers to the values:

Background: 00
Playfield 1: 02
Playfield 2: 0A
Playfield 3: 0C

Now put the following images into each of the screen RAM areas:

Pixel Contents (by Playfield Color Register)

First frame	1	1	1	1	2	3	2	3	2	3	2	3
Second frame	B	1	1	1	B	B	2	3	2	3	2	3
Third frame	B	B	1	1	B	B	B	B	2	3	2	3
Fourth frame	B	B	B	1	B	B	B	B	B	B	2	3
Effective luminance x4	2	4	6	8	10	12	20	24	30	36	40	48
Perceived luminance												

In this way, much finer luminance resolutions possible.

A final suggestion concerns a subject that is laden with opportunities but little understood as yet: the dynamic display list This is a display list which the 6502 changes during vertical blank periods. It should be possible to produce interesting effects with such lists For example, a text editing program dynamically inserts blank lines above and below the screen line being edited set it. apart from the other lines of text. As the cursor is moved vertically, the display list is changed. The technique is odd but very effective.

Chapter III

Graphics indirection

3 Graphics indirection

(COLOR REGISTERS AND CHARACTER SETS)
Indirection is a powerful concept in programming. In 6502 assembly language, there are three levels of indirection in referring to numbers. The first and most direct level is the immediate addressing mode In which the number itself is directly stated:

```
LDA #$F4
```

The second level of indirection is reached when the program refers to a memory location that holds the number:

```
LDA $0602
```

The third and highest level of indirection with the 6502 is attained when the program refers to a pair of memory locations which together contain the address of the memory location that holds the number. In the 6502, this indirection is complicated by the addition of an index:

```
LDA ($D0),Y
```

Indirection provides a greater degree of generality (and hence power) to the programmer. Instead of trucking out the same old numbers every time you want to get something done, you can simply point to them. By changing the pointer, you can change the behaviour of the program. Indirection is obviously an important capability.

COLOR REGISTERS
Graphics indirection is built into the ATARI Home Computer in two ways: with color registers and with character sets. Programmers first approaching this computer after programming other systems often think in terms of direct colors. A color register is a more complex beast than a color. A color specifies a permanent value. A color register is indirect; it holds any color value. The difference between the two is analogous to the difference between a box-end wrench and a socket wrench. The box-end wrench comes in one size only but a socket wrench can hold almost any size socket. A socket wrench is more flexible but takes a little more skill to use properly. Similarly, a color register is more flexible than a color but takes more skill to use effectively.

There are nine color registers in the ATARI 400/800 Computer; four are for player- missile graphics and will be discussed in Section 4. The remaining five are not always used; depending on the graphics mode used, as few as two registers or as many as five will show up on the screen. In BASIC mode 0, only two and one-half registers are used because the hue value of the characters is ignored; characters take the same hue as playfield register 2 but take their luminance from register 1. The color registers are in CTIA at addresses $D016 through $D01A. They are "shadowed" from OS RAM locations into CTIA during vertical blank. Figure 3-1 gives color register shadow and hardware addresses.

Image Controlled	Hardware		OS Shadow	
	Label	Address	Label	Address
Player 0	COLPM0	D012	PCOLR0	2C0
Player 1	COLPM1	D013	PCOLR1	2C1
Player 2	COLPM2	D014	PCOLR2	2C2
Player 3	COLPM3	D015	PCOLR3	2C3
Playfield 0	COLPF0	D016	COLOR0	2C4
Playfield 1	COLPF1	D017	COLOR1	2C5
Playfield 2	COLPF2	D018	COLOR2	2C6
Playfield 3	COLPF3	D019	COLOR3	2C7
Background	COLBK	D0lA	COLOR4	2C8

Figure 3-1 Color Register Labels and Addresses

For most purposes, the user controls the color registers by writing to the shadow locations. There are only two cases in which the programmer would write directly to the CTIA addresses. The first and most common is the display list interrupt which will be discussed in Section 5. The second

arises when the user disables the OS vertical blank interrupt routines that move the shadow values Into CTIA. Vertical blank interrupts are discussed in Section 8.

Colors are encoded in a color register by a simple formula. The upper nybble gives the hue value, which is identical to the second parameter of the BASIC SETCOLOR command. Table 9-3 of the BASIC Reference Manual lists hue values. The lower nybble in the color register gives the luminance value of the color. It is the same as the third parameter in the BASIC SETCOLOR command. The lowest order bit of this nybble is not significant. Thus, there are eight luminances for each hue. There are a total of 128 colors from which to choose (8 luminances times 16 hues). In this book, the term `color' denotes a hue- luminance combination.

Once a color is encoded into a color register, it is mapped onto the screen by referring to the color register that holds it. In map display modes which support four color registers the screen data specifies which color register is to be mapped onto the screen. Since there are four color registers it takes only two bits to encode one pixel. Thus, each screen data byte holds data for four pixels. The value in each pair of bits specifies which color register provides the color for that pixel.

In text display modes (BASIC's GRAPHICS modes 1 and 2) the selection of color registers is made by the top two bits of the character code. This leaves only six bits for defining the character, which is why these two modes have only 64 characters available.

Color register indirection gives you four special capabilities. First, you can choose from 128 different colors for your displays. This allows you to choose the color that most nearly meets your needs.

Second, you can manipulate the color registers in real time to produce pretty effects. The simplest version of this is demonstrated by the following BASIC line:

```
FOR I=0 TO 254 STEP 2:POKE 712,I:NEXT I
```

This line simply cycles the border color through all possible colors. The effect is quite pleasing and certainly grabs attention. The fundamental technique can be extended in a variety of ways. A special variation of this is to create simple cyclic animation by drawing a figure in four colors and then cycle the colors through the color registers rather than redrawing the figure. The following program illustrates the idea:

```
10 GRAPHICS 23
20 FOR X=0 TO 39
30 FOR I=0 TO 3
40 COLOR I
50 PLOT 4*X+I,O
60 DRAWTO 4*X+1,95
70 NEXT I
80 NEXT X
90 A=PEEK(712)
100 POKE 712,PEEK(710)
110 POKE 710,PEEK(709)
120 POKE 709,PEEK(708)
130 POKE 708,A
140 GOTO 90
```

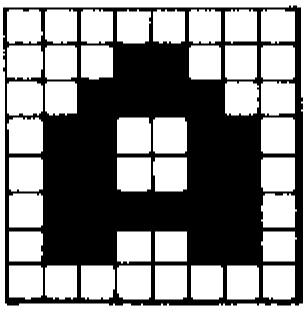
The third application of color registers is to logically key colors to situations. For example, a paged menu system can be made more understandable by changing the background color or the border color for each page in the menu. Perhaps the screen could flash red when an illegal key is pressed. The use of the color characters available in BASIC Graphics modes 1 and 2 can greatly extend the impact of textual material. An account sum could be shown in red if the account is in the red, or black if the account is in the black. Important words or phrases can be shown in special colors to make them stand out. The use of colors in map modes (no text) can also improve the utility of such graphics. A single graphics image (a monster, a boat, or whatever) could be presented in several different colors to represent several different versions of the same thing. It costs a great deal of RAM to store an image, but it costs very little to change the color of an

existing image. For example, it would be much easier to show three different boats by presenting one boat shape in three different colors than three different boat shapes.

The fourth and most important application of color registers Is used with display list interrupts. A single color register can be used to put up to 128 colors onto a single screen. This very Important capability will be discussed in Sect ton 5.

CHARACTER SETS
Graphics indirection is also provided through the use of redefinable character set. A standard character set is provided in ROM, but there is no reason why this particular character set must be used. The user can create and display any character set desired. There are three steps necessary to use a redefined character set. First, the programmer must define the character set. This is the most time-consuming step. Each character is displayed on the screen on an 8x8 grid; it is encoded in memory as an 8-byte table. Figure 3-2 depicts the encoding arrangement.

Character Image



Binary Representation	**Hex Representation**
00000000	00
00011000	18
00111100	3C
01100110	66
01100110	66
01111110	7E
01100110	66
00000000	00

Figure 3-2 Character Encoding

A full character set has 128 characters in it, each with a normal and an inverse video incarnation. Such a character set needs 1024 bytes of space and must start on a 1K boundary. Character sets for BASIC modes 1 and 2 have only 64 distinct characters, and so require only 512 bytes and must start on a 1/2K boundary. The first 8 bytes define the zeroth character, the next 8 bytes define the first character, and so on. Obviously, defining a new character set is a big job. Fortunately, there are software packages on the market to make this job easier.

Once the character set is defined and placed into RAM, you must tell ANTIC where it can find the character set. This is done by poking the page number of the beginning of the character table into location $D409 (decimal 54281). The OS shadow location, which is the location you would normally use, is called CHBAS and resides at $2F4 (decimal 756). The third step in using character sets is to print the character you want onto the screen. This can be done directly from BASIC with simple PRINT statements or by writing numbers directly into the screen memory.

A special capability of the system not supported in BASIC is the four- color character set option. BASIC Graphics modes 1 and 2 support five colors, but each character in these modes is really a two-color character; each one has a foreground color and a background color. The foreground color can be any of four single colors, but only one color at a time can be shown within a single character. This can be a serious hindrance when using character graphics.

There are two other text modes designed especially for character graphics. They are ANTIC modes 4 and 5. Each character in these modes is only four pixels wide, but each pixel can have four colors (counting background) The characters are defined just like BASIC Graphics mode 0 characters, except that each pixel is twice as wide and has two bits assigned to it to specify the color register used. Unlike ANTIC modes 6 and 7 (BASIC modes 1 and 2), color register selection is not made by the character name byte but instead by the defined character set. Each byte in the

character table is broken into four bit pairs, each of which selects the color for a pixel. (This is why there are only four horizontal pixels per character.) The highest bit (D7) of the character name byte modifies the color register used. Color register selection is made according to Figure 3-3:

bit pair in character defn	D7 = 0	D7 = 1
00	COLBAK	COLBAK
01	PFO	PFO
10	PF1	PF1
11	PF2	PF3

Figure 3-3 Color Register Selection for Characters

Using these text modes, multicolored graphics characters can be put onto the screen.

Another interesting ANTIC character mode is the lowercase descenders mode (ANTIC mode 3). This mode displays 10 scan lines per mode line, but since characters use only eight bytes vertically, the lower two scan lines are normally left empty. If a character in the last quarter of the character set is displayed, the top two scan lines of the character will be left empty; the data that should have been displayed there will instead be shown on the bottom two lines. This allows the user to create lowercase characters with descenders.

APPLICATIONS OF CHARACTER SETS

Many interesting and useful application possibilities spring from character set Indirection. The obvious application is the modified font. A different font can give a program a unique appearance. It is possible to have Greek, Cyrillic, or other special character sets. Going one step further, you can create graphics fonts. The ENERGY CZAR(TM) computer program uses a redefined character set for bar graphs. A character occupies eight pixels; tints means that bar charts implemented with standard characters have a resolution of eight pixels, a rather poor resolution. ENERGY CZAR uses a special character set in which some of the less popular text symbols (ampersands, pound signs, and the like) have been replaced with special bar chart characters. One character is a one-pixel bar, another is a two-pixel bar, and so on to the full eight-pixel bar. The program can thus draw detailed bar charts with resolution of a single pixel. Figure 3-4 shows a typical display from this program. The mix of text with map graphics is only apparent; the entire display is constructed with characters.

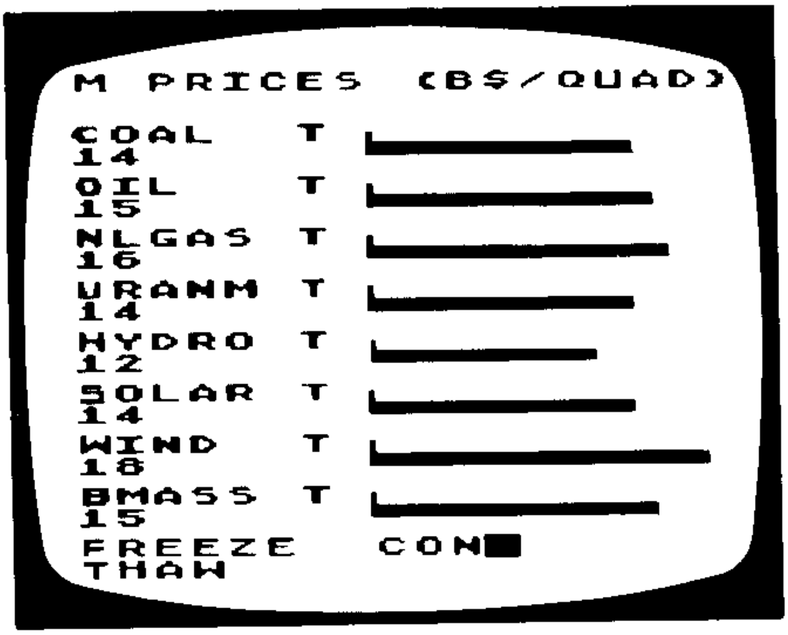


Figure 3-4 Energy CZAR Bar Charts

In many applications, character sets can be created that show special images. For example, by defining a terrain graphics character set with river characters, forest characters, mountain

characters, and so forth, It is possible to make a terrain map of any country. Indeed, with imagination a map of terrain on a different planet can be done just as easily. When doing this, it is best to define five to eight characters for each terrain type. Each variation of a single type should be positioned slightly differently in the character pixel. By mixing the different characters together, It is possible to avoid the monotonous look that Is characteristic of primitive character graphics. Most people won't realize that the resulting map uses character graphics until they study the map closely. Figure 3-5 shows a display of a terrain map created with character set graphics. The reproduction in black and white does not do justice to the original display, which has up to 18 colors.

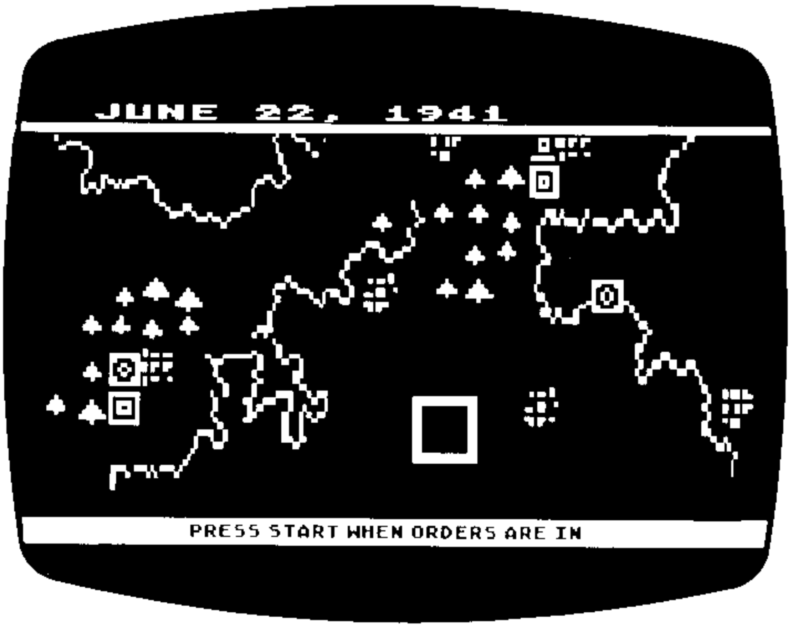


Figure 3-5 Terrain Map With Character Set Graphics

You could create an electronics character set with transistor characters, diode characters, wire characters, and so forth to produce an electronics schematics program. Or you could create an architectural character set with doorway characters, wall characters, corner characters, and so on to make an architectural blueprint program. The graphics possibilities opened up by character graphics with personal computers have not been fully explored.

Characters can be turned upside down by pokeing a 4 into location 755. One possible application of this feature might be for displaying playing cards (as in a Blackjack game). The upper half of the card can be shown right side up; with a display list interrupt the characters can be turned upside down for the lower half of the card. This feature might also be of some use in displaying images with mirror reflections (reflection pools, lakes, etc.

Even more exciting possibilities spring to mind when you realize that It is quite practical to change character sets while the program is running. A character set costs either 512 bytes or 1024 bytes; in either case It is quite inexpensive to keep multiple character sets in memory and flip between them during program execution. There are three time regimes for such character set multiplexing: human slow (more than 1 second); human fast (1/60 second to 1 second); and machine fast (faster than 1/ 60 sec).

Human-slow character set multiplexing is useful for "change of scenery" work. For example, a space travel program might use one graphics character set for one planet, another set for space, and a third set for another planet. As the traveller changes locations, the program changes the character set to give exotic new scenery. An adventure program might change character sets as the player changes locales.

Human-fast character set multiplexing is primarily of value for animation. This can be done in two ways: changing characters within a single character set, and changing whole character sets. The SPACE INVADERS (trademark of Taito America Corp.) program on the ATARI Home Computer

uses the former technique. The invaders are actually characters. By rapidly changing the characters, the programmer was able to animate them. This was easy because there are only six different monsters; each has four different Incarnations.

High-speed cyclic animation of an entire screen is possible by setting up a number of character sets, drawing the screen image, and then simply cycling through the character sets. If each character has a slightly different incarnation in each of the character sets, that character will go through an animated sequence as the character sets are changed. In this way a screen full of objects could be made to cyclically move with a very simple loop. Once the character set data is in place and the screen has been drawn, the code to animate the screen would be this simple:

```
1000 FOR 1=1 TO 10
1010 POKE 756,CHARBASE(I)
1020 NEXT I
1030 GOTO 1000
```

Computer-fast character set animation is used to put multiple character sets onto a single screen. This makes use of the display list interrupt capability of the computer. Display list interrupts are discussed in Sect ton 5.

The use of character sets for graphics and animation has many advantages and some limitations. The biggest advantage is that it costs very little RAM to produce detailed displays. A graphics display using BASIC mode 2 characters (such as the one shown in Figure 3-5) can give as much detail and one more color than a BASIC mode 7 display. Yet the character Image will cost 200 bytes while the map image will cost 4000 bytes. The RAM cost for multiple character sets is only 512 bytes per set, so It Is inexpensive to have multiple character sets. Screen manipulations with character graphics are much faster because you have less data to manipulate. However, character graphics are not as flexible as map graphics. You cannot put anything you want anywhere on the screen. This limitation would preclude the use of character graphics in some applications. However, there remain many graphics applications for which the program need display only a limited number of predefined shapes in fixed locations. In these cases, character graphics provide great utility.

Chapter IV

Player-Missile graphics

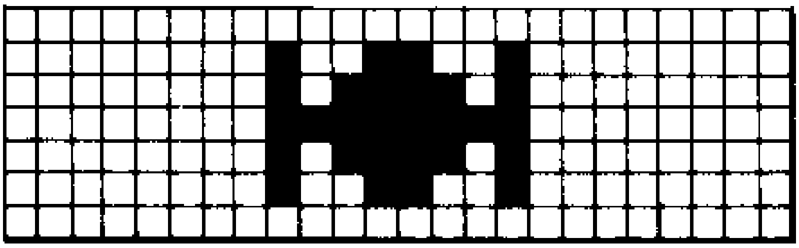
4 Player-Missile graphics

DIFFICULTIES WITH HIGH-SPEED ANIMATION
Animation is an important capability of any home computer system. Activity on the screen can greatly add to the excitement and realism of any program. Certainly animation is crucial to the appeal of many computer games. More importantly, an animated image can convey information with more impact and clarity than a static image. It can draw attention to an item or event of importance. It can directly show a dynamic process rather than indirectly talk about it. Animation must accordingly be regarded as an important element of the graphics capabilities of any computer system.

The conventional way to effect animation with home computers is to move the image data through the screen RAM area. This requires a two-step process. First, the program must erase the old image by writing background values to the RAM containing the current image. Then the program must write the image data to the RAM corresponding to the new position of the image. By repeating this process over and over, the image will appear to move on the screen.

There are problems with this technique. First, if the animation is being done in a graphics mode with large pixels, the motion will not be smooth; the image will jerk across the screen. With other computers the only solution is to use a graphics mode with smaller pixels (higher resolution). The second problem is much worse. The screen is a two-dimensional image, but the screen RAM is organized one-dimensionally. This means that an image which is contiguous on the screen will not be contiguous in the RAM. The discrepancy is illustrated in Figure 4-1.

Image



Correspondin g Bytes in RAM

00 00 00
00 99 00
00 BD 00
00 FF 00
00 BD 00
00 99 00
00 00 00

Spacing of Bytes in RAM:

00 00 00 00 99 00 00 BD 00 00 FF 00 00 BD 00 00 99 00 00 00 00

Image Bytes Scattered Through RAM

Figure 4-1 Noncontiguous RAM Images

The significance of this discrepancy does not become obvious until you try to write a program to move such an image. Look how the bytes that make up the image are scattered through the RAM. To erase them, your program must calculate their address. This calculation is not always easy to do. The assembly code just to access a single byte at screen location (XPOS, YPOS) would look like this (this code assumes 40 bytes per screen line):

```
LDA SCRNRM     Address of beginning of screen RAM
STA POINTR     zero page pointer
LDA SCRNRM+1   high order byte of address
STA POINTR+1   high order pointer
LDA #$00
STA TEMPA+1    temporary register
LDA YPOS       vertical position
ASL A          times 2
ROL TEMPA+1    shift carry into TEMPA+1
ASL A          times 4
ROL TEMPA+1    shift carry again
ASL A          times 8
```

```
ROL TEMPA+1    shift again
LDX TEMPA+1    save YPOS*8
STX TEMPB+1    into TEMPB
STA TEMPB      low byte
ASL A          times 16
ROL TEMPA+1
ASL A          times 32
ROL TEMPA+1
CLC
ADC TEMPB      add YPOS*8 to get YPOS*40
STA TEMPB
LDA TEMPA+1    now do high order byte
ADC TEMPB+1
STA TEMPB+1
LDA TEMPB      TEMPB contains the offset from the top of screen to pixel
CLC
ADC POINTR
STA POINTR
LDA TEMPB+1
ADC POINTR+1
STA POINTR+1
LDY XPOS
LDA (POINTR),Y
```

Clearly, this code to access a screen location is too cumbersome. This is certainly not the most elegant or fastest code to solve the problem. Certainly a good programmer could take advantage of special circumstances to make the code more compact. The point is that accessing pixels on a screen takes a lot of computing. The above routine takes about 100 machine cycles to access a single byte on the screen. To move an image that occupies, say, 50 bytes, would require 100 accesses or about 10,000 machine cycles or roughly 10 milliseconds. This may not sound like much, but if you want to achieve smooth motion, you have to move the object every 17 milliseconds. If there are other objects to move or any calculations to carry out there isn't much processor time left to devote to them. What this means is that this type of animation (called "playfield animation") is too slow for many purposes. You can still get animation this way, but you are limited to few objects or small objects or slow motion or few calculations between motion. The trade-offs that a programmer must make in using such animation are too restrictive.

PLAYER-MISSILE FUNDAMENTALS
The ATARI Home Computer solution to this problem is player-missile graphics. In order to understand player-missile graphics, it is important to understand the essence of the problem of playfield animation: the screen image is two-dimensional while the RAM image is one-dimensional. The solution was to create a graphics object that is one-dimensional on the screen as well as one-dimensional in RAM. This object (called a player) appears in RAM as a table that is either 128 or 256 bytes long. The table is mapped directly to the screen. It appears as a vertical band stretching from the top of the screen to the bottom. Each byte in the table is mapped into either one or two horizontal scan lines, with the choice between the two made by the programmer. The screen image is a simple bit-map of the data in the table. If a bit is on, then the corresponding pixel in the vertical column is lit; if the bit is off, then the corresponding pixel is off. Thus, the player image is not strictly one-dimensional; it is actually eight bits wide.

Drawing a player image on the screen is quite simple. First you draw a picture of the desired image on graph paper. The image must be no more than eight pixels wide. You then translate the image into binary code, substituting ones for illuminated pixels and zeros for empty ones. Then you translate the resulting binary number into decimal or hexadecimal, depending on which is more convenient. Then you store zeros into the player RAM to clear the image. Next, store the image data into the player RAM, with the byte at the top of the player image going first, followed by the other image bytes in top to bottom sequence. The further down in RAM you place you place data, the lower the image will appear on the screen.
VERTICAL MOTION
Animating this image is very easy. Vertical motion is obtained by moving the image data through the player RAM. This is, in principle, the same method used in playfield animation, but there is a big difference in practice; the move routine for vertical motion is a one-dimensional move instead of a two-dimensional move. The program does not need to multiply by 40 and it often does not need to use indirection. It could be as simple as:

```
     LDX $01
LOOP LDA PLAYER,X
```

```
STA PLAYER-1,X
INX
BNE LOOP
```

This routine takes about 4 milliseconds to move the entire player, about half as long as the playfield animation routine which actually moves only 50 bytes where this one moves 256 bytes. If high speed is necessary, the loop can be trimmed to move only the image bytes themselves rather than the whole player; then the loop would easily run in about 100-200 microseconds. The point here is that vertical motion with players is both simpler and faster than motion with playfield objects.

HORIZONTAL MOTION
Horizontal motion is even easier than vertical motion. There is a register for the player called the horizontal position register. The value in this register sets the horizontal position of the player on the screen. All you do is store a number into this register and the player jumps to that horizontal position. To move the player horizontally simply change the number stored in the horizontal position register. That's all there is to it.

Horizontal and vertical motion are independent; you can combine them in any fashion you choose.

The scale for the horizontal position register is one color clock per unit. Thus, adding one to the horizontal position register will move the player one color clock to the right. There are only 228 color clocks in a singe scan line; furthermore, some of these are not displayed because of overscan. The horizontal position register can hold 256 positions; some of these are off the left or right edge of the screen. Position 47 corresponds to the left edge of the standard playfield; position 208 corresponds to the right edge of the standard playfield. Thus, the visible region of the of the player is in horizontal positions 47 through 208. Remember, however, that this may vary from television to television due to differences in overscan. A conservative range of values is from 60 to 200. This coordinate range can sometimes be clumsy to use, but it does offer a nice feature: a simple way to remove a player from the screen is to set the player's horizontal position to zero. With a single load and store in assembly (or a singe POKE in BASIC), the player will disappear.

OTHER PLAYER-MISSILE FEATURES
The system described so far makes it possible to produce high-speed animation. There are a number of embellishments which greatly add to its overall utility. The first embellishment is that there are four individual players to use. These players all have their own sets of control registers and RAM area; thus their operation is completely independent. They are labelled P0 through P3. They can be used side by side to give up to 32 bits of horizontal resolution, or they can be used independently to give four movable objects.

Each player has its own color register; this color register is completely independent of the playfield color registers. The player color registers are called COLP(X) and are shadowed at PCOLR(X). This gives you the capability to put much more color onto the screen. However, each player has only one color; multicolored players are not possible without display list interrupts (display list interrupts are discussed in Section 5).

Each player has a controllable width; you can set it to have normal width, double width, or quadruple width with the SIZEP(X) registers. This is useful for making players take on different sizes. You also have the option of choosing the vertical resolution of the players. You can use single-line resolution, in which each byte in the player table occupies one horizontal scan line, or double-line resolution, in which each byte occupies two horizontal scan lines. With single-line resolution, each player bit-map table is 256 bytes long; with double-line resolution each table is 128 bytes long. This is the only case where player properties are not independent; the selection of vertical resolution applies to all players. Player vertical resolution is controlled by bit D4 of the DMACTL register. In single-line resolution, the first 32 bytes in the player table area lie above the standard playfield. The last 32 bytes lie below the standard playfield. In double-line resolution, 16 bytes lie above and 16 bytes lie below the standard playfield.

MISSILES
The next embellishment is the provision of missiles. These are 2-bit wide graphics objects associated with the players. There is one missile assigned to each player; it takes its color from the player's color register. Missile shape data comes from the missile bit-map table in RAM just in front

of the player's table. All four missiles are packed into the same table (four missiles times 2 bits per missile gives 8 bits). Missiles can move independently of players; they have their own horizontal position registers. Missiles have their own size register, SIZEM, which can set the horizontal width just like the SIZEP(X) registers do for players. However, missiles cannot be set to different sizes; they are all set together. Missiles are useful as bullets or for skinny vertical lines on the screen. If desired, the missiles can be grouped together into a fifth player, in which case they take the color of playfield color register 3. This is done by setting bit D4 of the priority control register (PRIOR). Note that missiles can still move independently when this option is in effect; their horizontal positions are set by their horizontal position registers. The fifth player enable bit only affects the color of the missiles.

You move a missile vertically the same way that you move a player: by moving the missile image data through the missile RAM area. This can be difficult to do because missiles are grouped into the same RAM table. To access a single missile, you must mask out the bits for the other missiles.

PLAYFIELD AND PLAYFIELD PRIORITIES
An important feature of player-missile graphics is that players and missiles are completely independent of the playfield. You can mix them with any graphics mode, text or map. This raises a problem: what happens if a player ends up on top of some playfield image? Which image has priority? You have the option to define the priorities used in displaying players. If you wish, all players can have priority over all playfield color registers. Or you can set all playfield color registers (except background) to have priority over all players. Or you can set player 0 and player 1 (henceforth referred to as P0 and P1) to have priority over all playfield color registers, with P2 and P3 having less priority than the playfield. Or you can set playfield color registers 0 and 1 (PF0 and PF1) the have priority over all players, which then have priority over PF2 and PF3. These priorities are selected with the priority control register (PRIOR) which is shadowed at GPRIOR. This capability allows a player to pass in front of one image and behind another, allowing three-dimensional effects.

HARDWARE COLLISION DETECTION
The final embellishment is the provision for hardware collision detection. This is primarily of value for games. You can check if any graphic object (player or missile) has collided with anything else. Specifically, you can check for missile-player collisions, missile-playfield collisions, player-player collisions, and player-playfield collisions. There are 54 possible collisions, and each one has a bit assigned to it that can be checked. If the bit is set, a collision has occurred. These bits are mapped into 15 registers in CTIA (only the lower 4 bits are used and some are not meaningful). These are read only registers; they cannot be cleared by writing zeros to them. The registers can be cleared for further collision detection by writing any value to register HITCLR. All collision registers are cleared by this command.

In hardware terms, a collision occurs when a player image coincides with another image; thus, the collision bit will not be set until the part of the screen showing the collision is drawn. This means that collision detection might not occur until as much as 16 milliseconds have elapsed since the player was moved. The preferred solution is to execute player motion and collision detection during the vertical blank interrupt routine (see Section 8 for a discussion of vertical blank interrupts). In this case, collision detection should be checked first, then collisions cleared, then players moved. Another solution is to wait at least 16 milliseconds after moving a player before checking for a collision involving that player.

There are a number of steps necessary to use player-missile graphics. First you must set aside a player-missile RAM area and tell the computer where it is. If you use single-line resolution, this RAM area will be 1280 bytes long; if you use double-line resolution it will be 640 bytes long. A good practice is to use the RAM area just in front of the display area at the top of RAM. The layout of the player-missile area is shown in Figure 4-2.

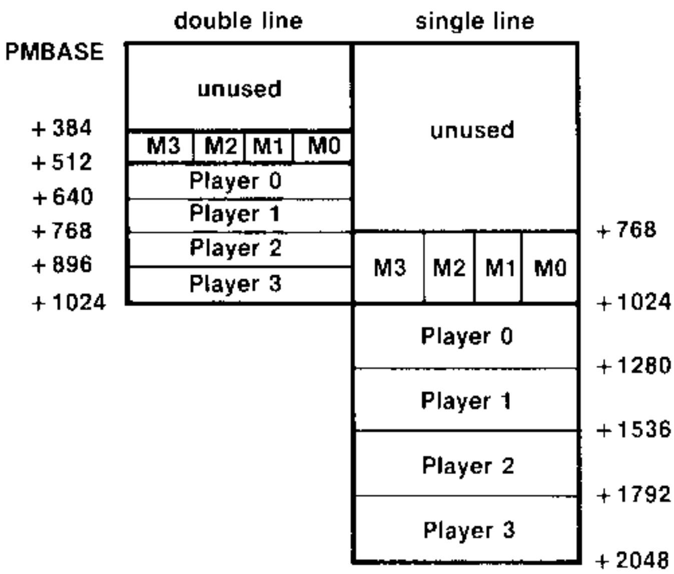


Figure 4-2 Player-Missile RAM Area Layout

The pointer to the beginning of the player-missile area is labelled PMBASE. Because of internal limitations of ANTIC, PMBASE must be on a 2K address boundary for single-line resolution, or a 1K address boundary for double-line resolution. If you elect not to use all of the players or none of the missiles, the areas of RAM set aside for the unused objects may be used for other purposes. Once you have decided where your player-missile RAM area will be, you inform ANTIC of this by storing the page number of PMBASE into the PMBASE register in ANTIC. Note that the address boundary restrictions on PMBASE preclude vertical motion of players by modifying PMBASE.

The next step is to clear the player and missile RAM by storing zeros into all locations in the player-missile RAM area. Then draw the players and missiles by storing image data into the appropriate locations in the player-missile RAM area.

Next, set the player parameters by setting the player color, horizontal position, and width registers to their initial values. If necessary, set the player/playfield priorities. Inform ANTIC of the vertical resolution you desire by setting bit D4 of register DMACTL (shadowed at SDMCTL) for single-line resolution, and clearing the bit for double-line resolution. Finally, enable the players by setting the PM DMA enable bit in DMACTL. Be careful not to disturb the other bits in DMACTL. A sample BASIC program for setting up a player and moving it with the joystick is given below:

```
1   PMBASE=54279:REM                      Player-missile base pointer
2   RAMTOP=106:REM                        OS top of RAM pointer
3   SDMCTL=559:REM                        RAM shadow of DMACTL register
4   GRACTL=53277:REM                      CTIA graphics control register
5   HPOSP0=53248:REM                      Horizontal position of P0
6   PCOLR0=704:REM                        Shadow of player 0 color
10  GRAPHICS 0: SETCOLOR 2,0,0:REM        Set background color to black
20  X=0:REM                               BASIC's player horizontal position
30  Y=48:REM                              BASIC's player vertical position
40  A=PEEK(RAMTOP)-8:REM                  Get RAM 2K below top of RAM
50  POKE PMBASE,A:REM                     Tell ANTIC where PM RAM is
60  MYPMBASE=256*A:REM                    Keep track of PM RAM address
70  POKE SDMCTL,46:REM                    Enable PM DMA with 2-line res
80  POKE GRACTL,3:REM                     Enable PM display
90  POKE HPOSP0,100:REM                   Declare horizontal position
100 FOR I=MYPMBASE+512 TO MYPMBASE+640:REM this loop clears player
110 POKE I,0
120 NEXT I
130 FOR I=MYPMBASE+512+Y to MYPMBASE+518+Y
140 READ A:REM                            This loop draws the player
150 POKE I,A
160 NEXT I
170 DATA 8,17,35,255,32,16,8
180 POKE PCOLR0,88:REM                    Make the player pink
```

```
190 A=STICK(0):REM                          Read joystick
200 IF A=15 THEN GOTO 190:REM               If inactive, try again
210 IF A=11 THEN X=X-1: POKE HPOSP0,X
220 IF A=7 THEN X=X+1: POKE HPOSP0,X
230 IF A<>13 THEN GOTO 280
240 FOR I=8 TO 0 STEP -1
250 POKE MYPMBASE+512+Y+I,PEEK(MYPMBASE+511+Y+I)
260 NEXT I
270 Y=Y+1
280 IF A<>14 THEN GOTO 190
290 FOR I=0 TO 8
300 POKE MYPMBASE+511+Y+I,PEEK(MYPMBASE+512+Y+I)
310 NEXT I
320 Y=Y-1
330 GOTO 190
```

Once players are displayed, they can be difficult to remove from the screen. This is because the procedure by which they are displayed involves several steps. First, ANTIC retrieves player-missile data from RAM (if such retrieval is enabled in DMACTL). Then ANTIC ships the player-missile data to CTIA (if such action is enabled in GRACTL). CTIA displays whatever is in its player and missile graphics registers (GRAFP0 through GRAFP3 and GRAFM). Many programmers attempt to turn off player-missile graphics by clearing the control bits in DMACTL and GRACTL. This only prevents ANTIC from sending new player-missile data to CTIA; the old data in the GRAF(X) registers will still be displayed. To completely clear the players the GRAF(X) registers must be cleared after the control bits in DMACTL and GRACTL have been cleared. A simpler solution is to leave the player up but set its horizontal position to zero. Of course, if this solution is used, ANTIC will continue to use DMA to retrieve player-missile data, wasting roughly 70,000 machine cycles per second.

APPLICATIONS OF PLAYER-MISSILE GRAPHICS
Player-missile graphics allow a number of very special capabilities. They are obviously of great value in animation. They do have limitations: there are only four players and each is only eight bits wide If you need more bits of horizontal resolution you can always fall back on playfield animation. But for high-speed animation or quick and dirty animation, player-missile graphics work very well.

It is possible to bypass ANTIC and write player-missile data directly into the player-missile graphics registers (GRAFP(X)) in CTIA. This gives the programmer more control over player-missile graphics. It also increases his responsibilities concomitantly. The programmer must maintain a bit map of player-missile data and move it into the graphics registers at the appropriate times. The 6502 must therefore be slaved to the screen drawing cycle. (See the discussion of kernels in Chapter 5.) This is a clumsy technique that offers minor improvements in return for major programming efforts. The programmer who bypasses the hardware power offered by ANTIC must make up for it with his own effort.

Players can also be used to produce apparent 3-dimensional motion. This is accomplished with the player width option. Each player is drawn with one of several bit maps. One bit map shows the player as 6 bits wide, and another shows the player in 8 bits. When the 6 bit player is drawn at normal resolution, it will be 6 color clocks wide. The next size step is achieved by going to double width with the 6 bit image; this will be 12 color clocks wide. The 8 bit image will be 16 color clocks wide. Similarly, going to quadruple width will produce images 24 and 32 color clocks wide. Thus, the image can grow in size from 6 color clocks to 32 color clocks wide. This technique is used very effectively in STAR RAIDERS. The Zylons there are two players with 16 bits, so the size transitions are even smoother.

Player-missile graphics offer many capabilities in addition to animation. Players are an excellent way to increase the amount of color in a display. The four additional color registers they provide allow four more colors on each line of the display. Of course, the 8-bit resolution does limit he range of their application. There is a way around this that can sometimes be used. Take a player at quadruple width and put it onto the screen. Then set the priorities so that the player has lower priority than a playfield color. Next, reverse that playfield color with background, so that the apparent background color of the screen is really a playfield color. The player disappears behind this new false background. Now cut a hole in the false background by drawing true background on it. The player will show up in front of the true background color, but only in the area where true background has been drawn. In this way the player can have more than eight bits of horizontal resolution. A sample program for doing this:

```
1  RAMTOP=106:REM               OS top of RAM pointer
2  PMBASE=54279:REM             ANTIC player-missile RAM pointer
3  SDMCTL=559:REM               Shadow of DMACTL
4  GRACTL=53277:REM             CTIA graphics control register
5  HPOSP0=53248:REM             Horizontal position register of P0
6  PCOLR0=704:REM               Shadow of player 0 color register
7  SIZEP0=53256:REM             Player width control register
8  GPRIOR=623:REM               Priority control register
10  GRAPHICS 7
20  SETCOLOR 4,8,4
30  SETCOLOR 2,0,0
40  COLOR 3
50  FOR Y=0 TO 79:REM           This loop fills the screen
60  PLOT 0,Y
70  DRAWTO 159,Y
80  NEXT Y
90  A=PEEK(RAMTOP)-20:REM       Must back up further for GR. 7
100 POKE PMBASE,A
110 MYPMBASE=256*A
120 POKE SDMCTL,46
130 POKE GRACTL,3
140 POKE HPOSP0,100
150 FOR I=MYPMBASE+512 TO MYPMBASE+640
160 POKE I,255:REM              Make player solid color
170 NEXT I
180 POKE PCOLR0,88
190 POKE SIZEP0,3:REM           Set player to quadruple width
200 POKE GPRIOR,4:REM           Set priority
210 COLOR 4
220 FOR Y=30 TO 40
230 PLOT Y+22,Y
240 DRAWTO Y+43,Y
250 NEXT Y
```

This program produces the following display:

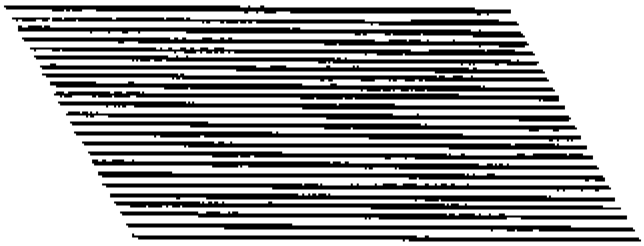


Figure 4-3 Masking a Player for More Resolution

SPECIAL CHARACTERS

Another application of player-missile graphics is for special characters. There are many special types of characters that cross vertical boundaries in normal character sets. One way to deal with these is to create special character sets that address this problem. Another way is to use a player. Subscripts, integral signs, and other special symbols can be done this way. A sample program for doing this is:

```
1   RAMTOP=106:REM                   OS top of RAM pointer
2   PMBASE=54279:REM                 ANTIC player-missile RAM pointer
```

```
3   SDMCTL=559:REM                       Shadow of DMACTL
4   GRACTL=53277:REM                     CTIA's graphics control register
5   HPOSP0=53248:REM                     Horizontal position register of P0
6   PCOLR0=704:REM                       Shadow of player 0 color register
10  GRAPHICS 0: A=PEEK(RAMTOP)-16:REM Must back up for 1-line resolution
20  POKE PMBASE,A
30  MYPMBASE=256*A
40  POKE SDMCTL,62
50  POKE GRACTL,3
60  POKE HPOSP0,102
70  FOR I=MYPMBASE+1024 TO MYPMBASE+1280
80  POKE I,0
90  NEXT I
100 POKE PCOLR0,140
110 FOR I=MYPMBASE+512+Y to MYPMBASE+518+Y
120 READ X
130 POKE MYPMBASE+1100+I,X
140 NEXT X
150 DATA 14,29,24,24,24,24,24,24
160 DATA 24,24,24,24,24,24,184,112
170 ?" ":REM                             Clear screen
180 POSITION 15,6
190 ?"xdx"
```

This program produces the following display:



Figure 4-4 Using a Player as a Special Character

A particularly useful application of players is for cursors. With their ability to smoothly move anywhere over the screen without disturbing its contents they are ideally suited for such applications. The cursor can change color as it moves over the screen to indicate what it has under it.

Player-missile graphics provide many capabilities. Their uses for action games as animated objects are obvious. They have many serious uses as well. They can add color and resolution to any display. They can present special characters. They can be used as cursors. Use them.

Chapter V

Display list interrupts

5 Display list interrupts

The display list interrupt (DLI) is one of the most powerful capabilities built into the ATARI Home Computer. It is also one of the least accessible features of the system, requiring a firm understanding of assembly language as well as all of the other characteristics of the machine. Display list interrupts all by themselves provide no additional capabilities; they must be used in conjunction with the other features of the system such as player-missile graphics, character set indirection, or color register indirection. With display list interrupts the full power of these features can be deployed.

THEORY OF OPERATION
Display list interrupts take advantage of the sequential nature of the raster scan television display. The television draws the screen image in a time sequence. It draws images from the top of the screen to the bottom. This drawing process takes about 17,000 microseconds, which looks instantaneous to the human eye, but is a long time in the time scale that the computer works in. The computer has plenty of time to change the parameters of the screen display while it is being drawn. Of course, it must effect each change each time the screen is drawn, which is 60 times per second. Also (and this is the tricky part), it must change the parameter in question at exactly the same time each time the screen is drawn. That is, the cycle of changing screen parameters must be synchronized to the screen drawing cycle. One way to do this might be to lock the 6502 up into a tight timing loop whose execution frequency is exactly 60 Hertz. This would make it very difficult to carry out any computations other than the screen display computations. It would also be a tedious job. A much better way would be to interrupt the 6502 just before the time has come to change the screen parameters. The 6502 responds to the interrupt, changes the screen parameters, and returns to its normal business. The interrupt to do this must be precisely timed to occur at exactly the same time during the screen drawing process. This specially timed interrupt is provided by the ANTIC chip; it is called a display list interrupt (DLI).

The timing and execution of any interrupt process can be intricate; therefore we will first narrate the sequence of events in a properly working DLI. The process begins when the ANTIC chip encounters a display list instruction with its interrupt bit (bit D7) set. ANTIC waits until the last scan line of the mode line it is currently displaying. ANTIC then refers to its NMIEN register to see if display list interrupts have been enabled. If the enable bit is clear, ANTIC ignores the interrupt and continues its regular tasks. If the enable bit is set, ANTIC pulls down the NMI line on the 6502. ANTIC then goes back to its normal display activities. The 6502 vectors through the NMI vector to an interrupt service routine in the OS. This routine first determines the cause of the interrupt. If the interrupt is indeed a DLI, the routine vectors through addresses $0200, $0201 (low then high) to a DLI service routine. The DLI routine changes one or more of the graphics registers which control the display. Then the 6502 RTIs to resume its mainline program.

There are a number of steps involved in setting up a DLI. The very first thing you must do is write the DLI routine itself. The routine must push any 6502 registers that will be altered onto the stack, as the OS interrupt poll routine saves no registers. (The 6502 does automatically push the Processor Status Register onto the stack.) The routine should be short and fast; it should only change registers related to the display. It should end by restoring any 6502 registers pushed onto the stack. Next you must place the DLI service routine somewhere in memory. Page 6 is an ideal place. Set the vector at $0200, $0201 to point to your routine. Determine the vertical point on the screen where you want the DLI to occur, then go to the corresponding display list instruction and set bit D7 of the previous instruction. Finally, enable the DLI by setting bit D7 of the NMIEN register at $D40E. The DLI will immediately begin functioning.
DLI TIMING
As with any interrupt service routine, timing considerations can be critical. ANTIC does not send the interrupt to the 6502 immediately upon encountering an interrupt instruction; it delays this until the last scan line of the interrupting mode line. There are a number of processing delays before the DLI reaches your service routine. Thus, your DLI service routine will begin executing while the electron beam is partway across the screen in the last scan line of the interrupting mode line. For example, if such a DLI routine changes a color register, the old color will be displayed on the left half of the screen and the new color will show up on the right half of the screen. Because of uncertain timing in the response of the 6502 to an interrupt, the border between them will not be sharp but will jiggle back and forth irritatingly.

There is a solution to this problem. It is provided in the form of the WSYNC (wait for horizontal sync) register. Whenever this register is addressed in any way, the ANTIC chip pulls down the ROY line on the 6502. This effectively freezes the 6502 until the register is reset by a horizontal sync. The effect is that the 6502 freezes until the electron beam reaches the right edge of the standard playfield. If you insert a STA WSYNC instruction just before the instruction which stores a value into a color register, the color will go into the color register while the beam is off the screen. The color transition will occur one scan line lower, but will be neat and clean.

The proper use of a DLI then is to set the DLI bit on the mode line before the mode line for which you want the action to occur. The DLI service routine should first save the 6502 registers onto the stack, and then load the 6502 registers with the new graphics values to be used. It should execute a STA WSYNC, and then store the new values into the appropriate ANTIC or CTIA registers. Finally, it should restore the 6502 registers and return from the interrupt. This procedure will guarantee that the graphics registers are changed at the beginning of the desired line while the electron beam is off the screen.

DLI EXAMPLE
A simple program demonstrating a DLI is given below:

```
10 DLIST=PEEK(560)+256*PEEK(561):REM   Find display list
20 POKE DLIST+15,130:REM               Insert interrupt instruction
30 FOR l=0 TO 19:REM                   Loop for poking DLI service routine
40 READ A:POKE 1536+1,A:NEXT I
50 DATA 72,138,72,169,80,162,88
60 DATA 141,10,212,141,23,208
70 DATA 141,24,208,104,170,104,64
80 POKE 512,0:POKE 513,6:REM           Poke in interrupt vector
90 POKE 54286, 192: REM                Enable DLI
```

This routine uses the following assembly language DLI service routine:

```
PHA           Save accumulator
TXA
PHA           Save X-register
LDA #$50      Dark color for characters
LOX #$58      Pink
STA WSYNC     Wait
STA COLPF1    Store color
STX COLPF2    Store color
PLA
TAX
PLA           Restore registers
RTI           Done
```

This is a very simple DLI routine. It changes the background color from blue to pink. It also changes the color of the characters so that they show up as dark against the pink background. You might wonder why the upper half of the screen remains blue even though the DLI routine keeps stuffing pink into the color register. The answer is that the OS vertical blank interrupt routine keeps stuffing blue into the color register during the vertical blank period. The blue color comes from the OS shadow register for that color register. Every hardware color register is shadowed out to a RAM location. You may already know about these shadow registers at locations 708 through 712. For most purposes you can change colors by poking values into the shadow registers. If you poke directly into the hardware registers, the OS shadow process will wipe out your poked color within a 60th of a second. For DLIs, however, you must store your new color values directly into the hardware registers. You can not use a DLI to set the color of the first displayed line of the screen; the OS takes care of that line for you. Use DLIs to change colors of lines below the first line.
ATTRACT MODE
By stuffing colors directly into the hardware registers, you create a new problem: you defeat the automatic attract mode. Attract mode is a feature provided by the operating system. After nine minutes without a keypress, the colors on the screen begin to cycle through random hues at lowered luminances. This ensures that a computer left unattended for several hours does not burn an image into the television screen. 1t is easy to build attract mode into a display list interrupt. Only two lines of assembly code need be inserted into the DLI routine:

```
Old           New
LDA NEWCOL    LDA NEWCOL
STA WSYNC     EOR COLRSH
```

```
     STA COLPF2      AND DRKMSK
                     STA WSYNC
                     STA COLPF2
```

DRKMSK and COLRSH are zero page locations ($4E and $4F) set up and updated by the OS during vertical blank interrupt. When attract mode is not in force, COLRSH takes a value of 00 and DRKMSK takes $FF. When attract mode is in force, COLRSH is given a new random value every 4 seconds and DRKMSK holds a value of $F6. Thus, COLRSH scrambles the color and DRKMSK lops off the highest luminance bit.

DETAILED TIMING CONSIDERATIONS

The implementation of attract mode in DLIs exacerbates an already difficult problem: the shortage of execution time during a DLI. A description of DLI timing will make the problem more obvious. DLI execution is broken into three phases: - Phase One covers the period from the beginning of the DLI to the STA WSYNC instruction. During Phase One the electron beam is drawing the last scan line of the interrupting mode line. - Phase Two covers the period from the STA WSYNC instruction to the appearance of the beam on the television screen. Phase Two corresponds to horizontal blank; all graphics changes should be made during Phase Two. - Phase Three covers the period from the appearance of the beam on the screen to the end of the DLI service routine. The timing of Phase Three is not critical.

One horizontal scan line takes 114 processor clock cycles of real time. A DLI reaches the 6502 on cycle number 8. The 6502 takes from 8 to 14 cycles to respond to the interrupt. The OS routine to service the interrupt and vector it on to the DLI service routine takes 11 machine cycles. During this time from 1 to 3 cycles wi11 be stolen for memory refresh DMA. Thus, the DLI service routine is not reached until from 28 to 36 clock cycles have elapsed. For planning purposes we must assume the worst case and program as if the DLI service routine is reached on cycle number 36. Furthermore, the

STA WSYNC instruction must be reached by cycle number 100; this reduces the time available in Phase One by 14 cycles. Finally, ANTIC's DMA will steal some of the remaining clock cycles from the 6502. Nine cycles will be lost to memory refresh DMA. This leaves an absolute maximum of 55 cycles available for Phase One. This maximum is achieved only with blank line mode lines. Character and map mode instructions will result in the loss of one cycle for each byte of display data. The worst case arises with BASIC modes 0, 7, and 8, which require 40 bytes per line. Only 15 machine cycles are available to Phase One in such modes. Thus, a Phase One routine will have from 15 to 55 machine cycles of execution time available to it.

Phase Two, the critical phase, extends over 27 clock cycles of real time. As with Phase One, some of these cycles are lost to cycle stealing DMA. Player-missile graphics will cost five cycles if they are used. The display instruction will cost one cycle; if the LMS option is used, two more cycles will be stolen. Finally, one or two cycles may be lost to memory refresh or display data retrieval. Thus, from 17 to 26 machine cycles are available to Phase Two.

The problems of DLI timing now become obvious. To load, attract and store a single color will consume 14 cycles. Saving A, X, and Y onto the stack and then loading, attracting, and saving three colors into A, X, and Y will cost 47 cycles, most if not all of Phase One. Obviously, the programmer who wishes to use DLI for extensive graphics changes will expend much effort on the timing of the DLI. Fortunately, the beginning programmer need not be concerned with extensive timing calculations. If only single color changes or simple graphics operations are to be performed, cycle counting and speed optimization are unnecessary. These considerations are only important for high-performance situations.

There are no simple options for the programmer who needs to change more than three color registers in a single DLI. It might be possible to load, attract, and store a fourth color early in Phase Three if that color is not displayed on the left edge of the screen. Similarly, a color not showing up on the right side of the screen could be changed during Phase One. Another approach is to break one overactive DLI into two less ambitious DLIs, each doing half the work of the original. The second DLI could be provided by inserting a single scan line blank instruction (with DLI bit set) into the display list just below the main interrupting mode line. This will consume some screen space.

Another partial solution is to perform the attract chores during vertical blank periods. To do this, two tables of colors must be kept in RAM. The first table contains color values intended to be

displayed by the DLI routines. The second table contains the attracted values of these colors. During vertical blank, a user-supplied interrupt service routine fetches each color from the first table, attracts it, and stores the attracted color to the second table. The DLI routine then retrieves values directly from the second table without paying the time penalty for attract.

MULTIPLE DLIs

It is often desirable to have a number of DLIs occurring at several vertical positions on the screen. This is an important way to add color to a display. Unfortunately, there is only one DLI vector; if multiple DLIs are to be implemented then the vectoring to the appropriate DLI must be implemented, in the DLI routine itself. There are several ways to do this. If the DLI routine does the same process with different values then it can be table-driven. On each pass through the DLI routine, a counter is incremented and used as an index to a table of values. A sample DLI routine for doing this is as follows:

```
PHA
TXA
PHA
INC COUNTR
LDX COUNTR
LDA COLTAB,X          Use page two for color table
STA WSYNC             Wait
STA COLBAK
CPX #$4F              Last line?
BNE ENDDLI            No, exit
LDA #$00              Yes, reset counter
STA COUNTR
ENDDLI PLA
TAX
PLA                   Restore accumulator
RTI
```

The BASIC program to call this routine is:

```
10 GRAPHICS 7
20 DLIST=PEEK(560)+256*PEEK(561):REM       Find display list
30 FOR J=6 TO 84:REM                        Give every mode line a DLI
40 POKE DLIST+J,141:REM                     BASIC mode 7 with DLI bit set
50 NEXT J
60 FOR J=0 TO 30
70 READ A:POKE 1536+J,A:NEXTJ:REM           Poke in DLI service routine
80 DATA 72,138,72,238,32,6,175,32,6
90 DATA 189,0,240,141,10,212,141,26,208
100 DATA 224,79,208,5,169,0
110 DATA 141,32,6,104,170,104,64
120 POKE 512,0:POKE 513,6:REM               Vector to DLI service routine
130 POKE 54286,192:REM                      Enable DLI
```

This program will put 80 different colors onto the screen.

There are other ways to implement multiple DLIs. One way is to use a DLI counter as a test for branching through the DLI service routines to the proper DLI service routine. This slows down the response of all the DLIs, particularly the ones at the end of the test sequence. A better way is to have each DLI service routine write the address of the next routine into the DLI vector at $200, $201. This should be done during Phase Three. This is the most general solution to the problem of multiple DLIs. It has the additional advantage that vectoring logic is performed after the time critical portion of the DLI, not before.

The OS keyboard click routine interferes with the function of the DLI. Whenever a key is pressed and acknowledged, the onboard speaker is clicked. The timing for this click is provided by several STA WSYNC instructions. This can throw off the timing of a DLI routine and cause the screen colors to jump downward by one scan line for a fraction of a second. There is no easy solution to this problem. One possible solution involves the VCOUNT register, a read-only register in ANTIC which tells what scan line ANTIC is displaying. A DLI routine could examine this register to decide when to change a color. Another solution is to disable the OS keyboard service routine and provide your own keyboard routine. This would be a tedious job. The final solution is to accept no inputs from the keyboard. If key presses are-not acknowledged, the screen jiggle does not occur.

KERNELS

The DLI was designed to replace a more primitive software/hardware technique called a kernel. A kernel is a 6502 program loop which is precisely timed to the display cycle of the television set. By monitoring the VCOUNT register and consulting a table of screen changes catalogued as a function of VCOUNT values, the 6502 can arbitrarily control all graphics values for the entire screen. A high price is paid for this power: the 6502 is not available for computations during the screen display time, which is about 75 percent of the time. Furthermore, no computation may consume more than the 4000 or so machine cycles available during vertical blank and overscan periods. This restriction means that kernels can only be used with programs requiring little computation, such as certain skill and action games. For example, the BASKETBALL program for the ATARI 400/800 Computers uses a kernel; the program requires little computation but much color. The multicolored players in this game could not be done with display list interrupts, because DLIs are keyed to playfield vertical positions, not player positions.

It is possible to extend the kernel idea right into a single scan line and change graphics registers on the fly. In this way a single color register can present several colors on a single scan line. The horizontal position of the color change is determined by the amount of time that elapses before the change goes in. Thus, by carefully counting machine cycles, the programmer can get more graphics onto the screen. Unfortunately, this is extremely difficult to achieve in practice. With ANTIC DMAing the 6502, it is very difficult to know exactly how many cycles have really elapsed; a simple count of 6502 cycles is not adequate. If ANTIC's DMA Is turned off, the 6502 can assume full control of the display but must then perform all the work that ANTIC normally does. For these reasons horizontal kernels are seldom worth the effort. However, if the two images to be displayed in different colors are widely separated, say by 20 color clocks or more, the separation should cover up the timing uncertainties and render this technique feasible.

APPLICATIONS OF DISPLAY LIST INTERRUPTS

The tremendous value of graphics indirection and all those modifiable registers in the hardware now becomes obvious. With display list interrupts, every one of those registers can be changed on the fly. You can put lots of color, graphics, and special effects onto the screen. The most obvious application of DLIs is to put more color onto the screen. Each color register can be changed as many times as you have DLIs. This applies to both playfield color registers and player color registers. Thus, you have up to nine color registers, each of which can display up to 128 different colors. Is that enough color for you? Of course, a normal program would not lend itself to effectively using all of those colors. Too many DLIs start slowing down the whole program. Sometimes the screen layout cannot accommodate lots of DLIs. In practice, a dozen colors is easy, two dozen requires careful planning, and more than that requires a contrived situation.

Display list interrupts can give more than color; they can also be used to extend the power of player-missile graphics. The horizontal position of a player can be changed by a DLI. In this way a player can be repositioned partway down the screen. A single player can have several incarnations on the screen. If you imagine a player as a vertical column with images drawn on it, a DLI becomes a pair of scissors with which you can snip the column and reposition sections of it on the screen. Of course, no two sections of the player can be on the same horizontal line, so two incarnations of the player cannot be on the same horizontal line. If your display needs allow graphics objects that will never be on the same horizontal line, a single player can do the job.

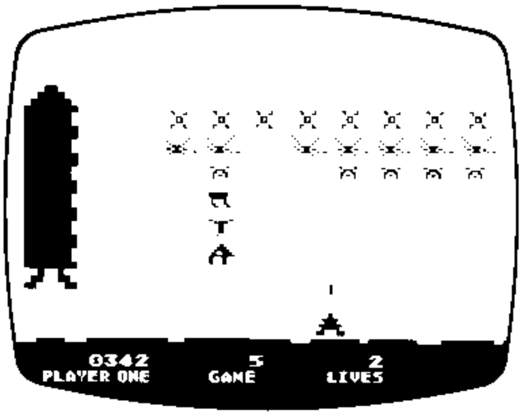
Another way to use DLIs in conjunction with players is to change their width or priority. This would most often be used along with the priority masking trick described in Section 4.

The last application of DLIs is the changing of character sets partway down the screen. This allows a program to use character graphics in a large window and regular text in a text window. Multiple character set changes are possible; a program might use one graphics character set at the top of the screen, another graphics character set in the middle of the screen, and a regular text character set at the bottom. A `Rosetta Stone' program would also be possible, showing different text fonts on the same screen. The vertical reflect bit can be changed with a DLI routine, allowing some text to be rightside up and other text to be upside down.

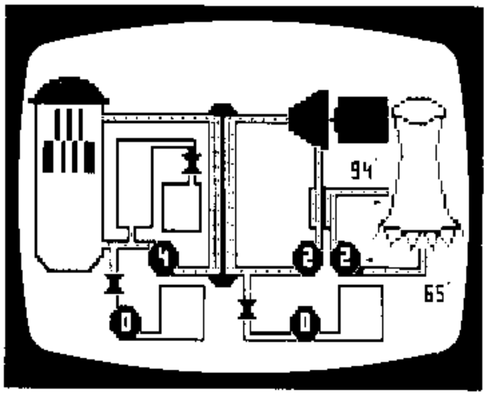
The proper use of the DLI requires careful layout of the screen display. The designer must give close consideration to the vertical architecture of display. The raster scan television system is not two-dimensionally symmetric; it has far more vertical structure than horizontal structure. This is because the pace for horizontal screen drawing is 262 times faster than the pace for vertical

screen drawing. The ATARI Home Computer display system was designed specifically for raster scan television, and it mirrors the anisotropy of the raster scan system. The ATARI Home Computer display is not a flat, blank sheet of paper on which you draw; it is a stack of thin strips, each of which can take different parameters. The programmer who insists on designing an isotropic display wastes many opportunities. You will achieve optimal results when you organize the information you wish to display in a strong vertical structure. This allows the full power of the DLI to be brought to bear.

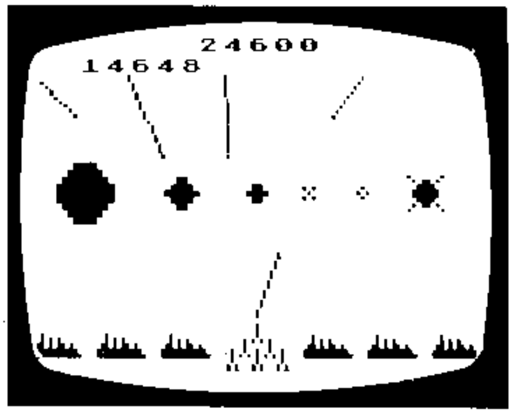
Figure 5-1 shows some screen displays from various programs and gives estimates of the degree of vertical screen architecture used in each.



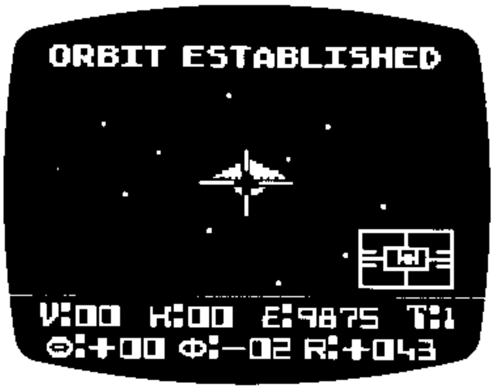
SPACE INVADERS
(Trademark of Taito America Corporation)
LOTS



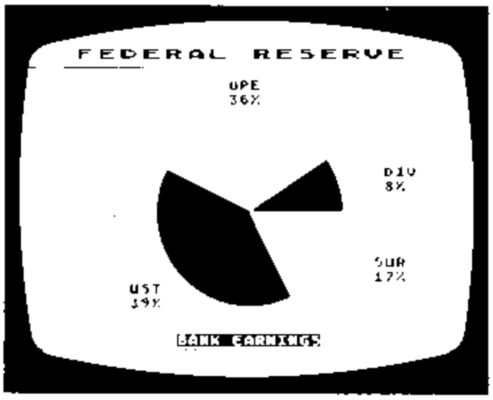
SCRAM
(A Nuclear Reactor Simulation)
LITTLE

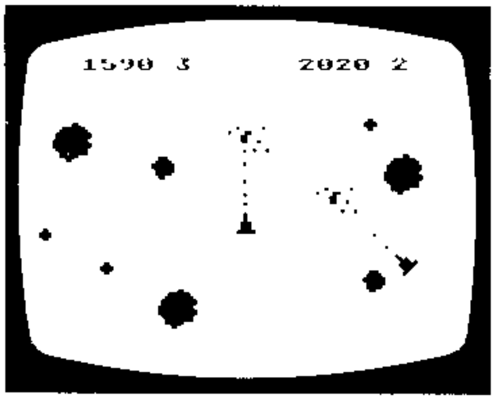


MISSILE COMMAND
SOME



STAR RAIDERS
LITTLE





GRAPH IT
NONE

ASTEROIDS
NONE

Figure 5-1 Examples of Vertical Screen Architecture

Chapter VI

Scrolling

6 Scrolling

SCROLLING
Quite frequently the amount of information that a programmer wants to display exceeds the amount of information that can fit onto the screen. One way of solving this problem is to scroll the information across the display. For example, listings of BASIC programs scroll vertically from the bottom to the top of the screen. All personal computers implement this type of scrolling. However, the ATARI Home Computer has two additional scrolling facilities that offer exciting possibilities. The first is "Load Memory Scan" (LMS) coarse scrolling; the second is fine scrolling.

Conventional computers use coarse scrolling; in this type of scrolling, the pixels that hold the characters are fixed in position on the screen and text is scrolled by moving bytes through the screen RAM. The resolution of the scrolling is a single character pixel, which is very coarse. The scrolling this produces is jerky and quite unpleasant. Furthermore, it is achieved by moving up to a thousand bytes around in memory, a slow and clumsy task. In essence, the program must move data through the playfield to scroll.

Some personal computers can produce a somewhat finer scroll by drawing images in a higher resolution graphics mode and then scrolling these images. Although higher scrolling resolution is achieved, more data must be moved to attain the scrolling and the program is consequently slowed. The fundamental problem is that the scrolling is implemented by moving data through the screen area.

There is a better way to achieve coarse scrolling with the ATAR.1 400/ 800: move the screen area over the data. The display list opcodes support the Load Memory scan feature. The LMS instruction was first described in Section 2 and tells ANTIC where the screen memory is. A normal display list will have one LMS instruction at the beginning of the display list; the RAM area it points to provides the screen data for the entire screen in a linear sequence. By manipulating the operand bytes of the LMS instruction., a primitive scroll can be Implemented. In effect, this moves the playfield window over the screen data. Thus, by manipulating just 2 address bytes, you can produce an effect identical to moving the entire screen RAM. The following program does just that:

```
10 DLIST=PEEK(560)+256*PEEK(561):REM Find display list
20 LMSLOW=DLIST+4:REM Get low address of LMS operand
30 LMSHIGH=DLIST+5:REM Get high address of LMS operand
40 FOR 1=0 TO 255:REM Outer loop
50 POKE LMSHIGH,
60 FOR J=O TO 255:REM Inner loop
70 POKE LMSLOW,J
80 FOR Y=1 TO 50:NEXT Y:REM Delay loop
90 NEXT J
100 NEXT 1
```

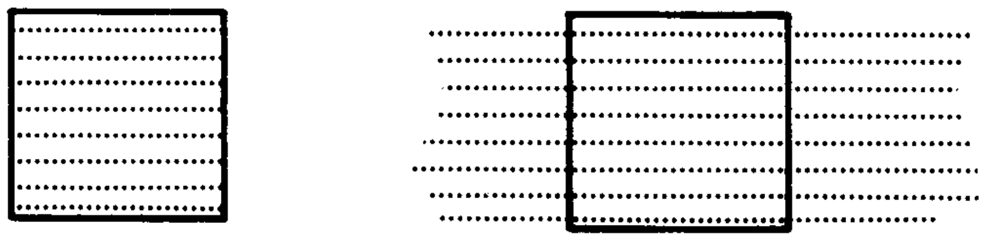
This program sweeps the display over the entire address space of the computer. The contents of the memory are all dumped onto the screen. The scroll is a clumsy serial scroll combining horizontal scrolling with vertical scrolling. A pure vertical scroll can be achieved by adding or subtracting a fixed amount (the line length in bytes) to the LMS operand. The following program does that:

```
10 GRAPHICS 0
20 DLIST=PEEK(560)+256*PEEK(561)
30 LMSLOW=DLIST+4
40 LMSHIGH=DLIST+5
50 SCREENLOW=0
60 SCREENHIGH=0
70 SCREENLOW=SCREENLOW+40:REM Next line
80 IF SCREENLOW<256 THEN GOTO 120:REM Overflow?
90 SCREENLOW=SCREENLOW-256:REM Yes, adjust pointer
100 SCREENHIGH=SCREENHIGH+1
110 IF SCREENHIGH=256 THEN END
120 POKE LMSLOW,SCREENLOW
130 POKE LMSHIGH,SCREENHIGH
140 GOTO 70
```

HORIZONTAL SCROLLING
A pure horizontal scroll is not so simple to do as a pure vertical scroll. The problem is that the screen RAM for a simple display list is organIzed serially. The screen data bytes for the lines are

strung in sequence, with the bytes for one line immediately following the bytes for the previous line. You can horizontally scroll the lines by shifting all the bytes to the left; this is done by decrementing the LMS operand. However, the leftmost byte on each line will then be scrolled into the rightmost position in the next higher line. The first sample program above illustrated this problem.

The solution is to expand the screen data area and break it up into a series of independent horizontal line data areas. Figure 6-1 schematically illustrates this idea:



normal data arrangement   arrangement for horizontal scroll

Figure 6-1 Arranging Screen RAM

On the left is the normal arrangement. One-dimensional serial RAM is stacked in linear sequence to create the screen data area. On the right is the arrangement we need for proper horizontal scrolling. The RAM is of course still one-dimensional and still serial, but now it is used differently. The RAM for each horizontal line extends much further than the screen can show. This is no accident; the whole point of scrolling is to allow a program to display more information than the screen can hold. You can't show all that extra information if you don't allocate the RAM to hold it. With this arrangement you can implement true horizontal scrolling. You can move the screen window over the screen data without the undesirable vertical roll of the earlier approach.

The first step in implementing pure horizontal scroll is to determine the total horizontal line length and allocate RAM accordingly. Next, you must write a completely new display list with an LMS instruction on each mode line. The display list will of course be longer than usual, but there is no reason why you cannot write such a display list. What values do you use for the LMS operands? It is most convenient to use the address of the first byte of each horizontal screen data line. There will be one such address for each mode line on the screen. Once the new display list is in place, ANTIC must be turned onto it and screen data must be written to populate the screen. To execute a scroll, each and every LMS operand in the display list must be incremented for a rightward scroll or decremented for a leftward scroll. Program logic must ensure that the image does not scroll beyond the limits of the allocated RAM areas; otherwise, garbage displays will result. In setting up such logic, the programmer must remember that the LMS operand points to the first screen data byte in the displayed line. The maximum value of the LMS operand is equal to the address of the last byte in the long horizontal line minus the number of bytes in one displayed line. Remember also that the LMS value should not come within one screen display line's length (in bytes) of a 4K address boundary, or the wrong data will be displayed due to LMS counter rollover.

As this process is rather intricate, let us work out an example. First, we must select our total horizontal line length. We shall use a horizontal line length of 256 bytes, as this will simplify address calculations. Each horizontal line will then require one page of RAM. Since we will use BASIC mode 2, there will be 12 mode lines on screen; thus, 12 pages or 3K of RAM will be required. For simplicity (and to guarantee that our screen RAM will be populated with nonzero data), we will use the bottom 3K of RAM. This area is used by the OS and DOS and so should be full of interesting data. To make matters more interesting, we'll put the display list onto page 6 so that we can display the display list on the screen as we are scrolling. The initial values of the LMS operands will thus be particularly easy to calculate; the low order bytes will all be zeros and the high order bytes will be (in order) 0, 1, 2, etc. The following program performs all these operations and scrolls the screen horizontally:

```
10 REM first set up the display list
20 POKE 1536,112:REM 8 blank lines
```

```
30 POKE 1537,112:REM 8 blank lines
40 POKE 1538,112:REM 8 blank lines
50 FOR 1=1 TO 12:REM Loop to put in display list
60 POKE 1536+3*1,71:REM BASIC mode 2 with LMS set
70 POKE 1536+3*1+1,0:REM Low byte of LMS operand
80 POKE 1536+3*1+2,1:REM High byte of LMS operand
90 NEXT I
100 POKE 1575,65:REM ANTIC JVB instruction
110 POKE 1576,0:REM Display list starts at $0600
120 POKE 1577,6
130 REM tell ANTIC where display list is
140 POKE 560.0
150 POKE 561,6
160 REM now scroll horizontally
170 FOR 1=0 TO 235:REM Loop through LMS low bytes
175 REM we use 235 --- not 255 --- because screen width is 20 characters
180 FOR J=1 TO 12:REM for each mode line
190 POKE 1536+3*J+1,1:REM Put in new LMS low byte
200 NEXT J
210 NEXT 1
220 GOTO 170:REM Endless loop
```

This program scrolls the data from right to left. When the end of a page is reached, it simply starts over at the beginning. The display list can be found on the sixth line down (it's on page 6). 1t appears as a sequence of double quotation marks.

The next step is to mix vertical and horizontal scrolling to get diagonal scrolling. Horizontal scrolling is achieved by adding 1 to or subtracting 1 from the LMS operand. Vertical scrolling is achieved by adding the line length to or subtracting the line length from the LMS operand. Diagonal scrolling is achieved by executing both operations. There are four possible diagonal scroll directions. If, for example, the line length is 256 bytes and we wish to scrolldown and to the right, we must add 256+(-1)=255 to each LMS operand in the display list. This is a 2-byte add; the BASIC program example given above avoids the difficulties of 2-byte address manipulations but most programs will not be so contrived. For truly fast two-dimensional scrolling, assembly language will be necessary.

All sorts of weird arrangements are possible if we differentially manipulate the LMS bytes. Lines could scroll relative to each other or hop over each other. Of course, some of this could be done with a conventional display but more data would have to be moved to do it. The real advantage of LMS scrolling is its speed. Instead of manipulating an entire screenfull of data, many thousands of bytes in size, a program need only manipulate two or perhaps a few dozen bytes.

FINE SCROLLING
The second important scrolling facility of the ATARI Computer is the fine scrolling capability. Fine scrolling is the capability of scrolling a pixel in steps smaller than the pixel size. (Throughout this section the term pixel refers to an entire character, not to the smaller dots that make up a character.) Coarse scrolls proceed in steps equal to one pixel dimension; fine scrolls proceed in steps of one scan line vertically and one color clock horizontally. Fine scrolling can only carry so far; to get full fine scrolling over long distances on the screen you must couple fine scrolling with coarse scrolling.

There are only two steps to implement fine scrolling. First, you set the fine scroll enable bits in the display list instruction bytes for the mode lines in which you want fine scrolling. (In most cases you want the entire screen to scroll so you set all the scroll enable bits in all the display list instruction bytes.) Bit D5 of the display list instruction is the vertical scroll enable bit; bit D4 of the display list instruction is the horizontal scroll enable bit. You then store the scrolling value you desire into the appropriate scrolling register. There are two scrolling registers, one for horizontal scrolling and one for vertical scrolling. The horizontal scroll register (HSCROL) is at $D404; the vertical scroll `register (VSCROL) is at $D405. For horizontal scrolling, you store into HSCROL the number of color clocks by which you want the mode line scrolled. For vertical scrolling, you store into VSCROL the number of scan lines that you want the mode line scrolled. These scroll values will be applied to every line for which the respective fine scroll is enabled.

There are two complicating factors that you encounter when you use fine scrolling. Both arise from the fact that a partially scrolled display shows more information than a normal display. Consider for example what happens when you horizontally scroll a line by half a character to the left. There are 40 characters in the line. Half of the first character disappears off of the left edge of the screen.

The 40th character scrolls to the left. What takes its place? Half of a new character should scroll in to take the place of the now scrolled 40th character. This character would be the 41st character. But there are only 40 characters in a normal line. What happens?

If you have implemented coarse scrolling, then the 41st character suddenly appears on the screen after the first character disappears off of the left edge. This sudden appearance is jerky and unsightly. The solution to this problem has already been built into the hardware. There are three display options for line widths: the narrow playfield (128 color clocks wide), the normal playfield (160 color clocks wide) and the wide playfield (192 color clocks wide). These options are set by setting appropriate bits in the DMACTL register. When using horizontal fine scrolling, ANTIC automatically retrieves more data from RAM than It displays. For example, if DMACTL is set for normal playfield, which in BASIC mode 0 has 40 bytes per line, then ANTIC will actually retrieve data at a rate appropriate to wide playfield --- 48 bytes per line. This will throw lines off horizontally if it is not taken into account. The problem does not manifest itself if the you have already organized screen RAM into long horizontal lines as in Figure 6-1.

The corresponding problem for vertical scrolling can be handled in either of two ways. The sloppy way is to ignore it. Then you will not get half-images at both ends of the display. Instead, the images at the bottom of the display will not scroll in properly; they will suddenly pop into view. The proper way takes very little work. To get proper fine scrolling into and out of the display region you must dedicate one mode line to act as a buffer. You do this by refraining from setting the vertical scroll bit in the display list instruction of the last mode line of the vertically scrolled zone. The window will now scroll without the unpleasant jerk. The screen image will be shortened by one mode line. An advantage of scrolling displays now becomes apparent. It is quite possible to create screen images that have more than 192 scan lines in the display. This could be disastrous with a static display, but with a scrolling display images which are above or below the displayed region can always be scrolled into view.

Fine scrolling will only scroll so far. The vertical limit for fine scrolling is 16 scan lines; the horizontal limit for fine scrolling is 16 color clocks. If you attempt to scroll beyond these limits, ANTIC simply ignores the higher bits of the scroll registers. To get full fine scrolling (in which the entire screen smoothly scrolls as far as you wish) you must couple fine scrolling with coarse scrolling. To do this, first fine scroll the image, keeping track of how far it has been scrolled. When the amount of fine scrolling equals the size of the pixel, reset the fine scroll register to zero and execute a coarse scroll. Figure 6-2 illustrates the process.

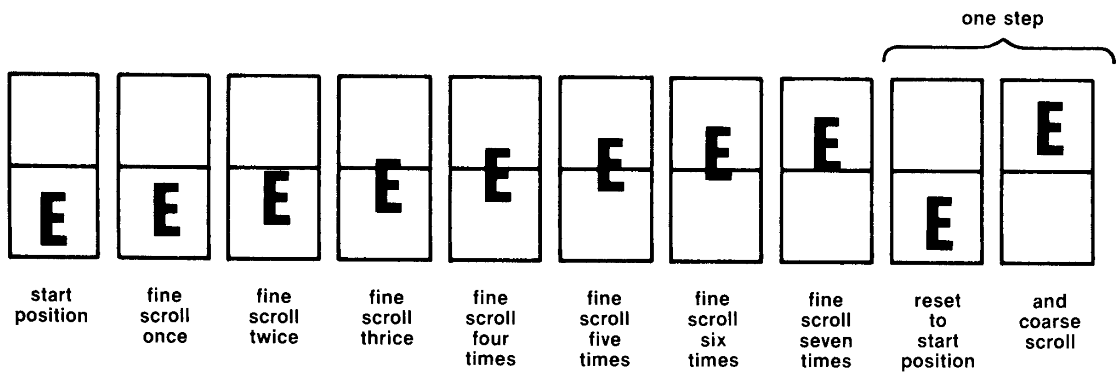


Figure 6-2 Linking Fine Scroll to Coarse Scroll

The following program illustrates simple fine scrolling:

```
1 HSCROL=54276
2 VSCROL=54277
10 GRAPHICS 0:LIST
20 DLIST=PEEK(560)+256*PEEK(561)
30 POKE DLIST+10,50:REM Enable both scrolls
40 POKE DLIST+11,50:REM Do it for two mode lines
50 FOR Y=O TO 7
60 POKE VSCROL,Y:REM Vertical scroll
```

```
70 GOSUB 200:REM Delay
80 NEXT Y
90 FOR X=0 TO 3
100 POKE HSCROL,X:REM Horizontal scroll
110 GOSUB 200:REM Delay
120 NEXT X
130 GOTO 40
200 FOR J=1 TO 200
210 NEXT J:RETURN
```

This program shows fine scrolling taking place at very slow speed. it demonstrates several problems that arise when using fine scrolling. First, the display lines below the scrolled window are shifted to the right. This is due to ANTIC's automatically retrieving 48 bytes per line instead of 40. The problem arises only in unrealistic demonstration programs such as this one. In real scrolling applications, the arrangement of the screen data (as shown in Figure 6-1) precludes this problem. The second, more serious problem arises when the scroll registers are modified while- ANTIC is in the middle of its display process. This confuses ANTIC and causes the screen to jerk. The solution is to change the scroll registers only during vertical blank periods. This can only be done with assembly language routines. Thus, fine scrolling normally requires the use of assembly language.

APPLICATIONS OF SCROLLING
The applications of full fine scrolling for graphics are numerous. The obvious application is for large maps that are created with character graphics. Using BASIC Graphics mode 2, 1 have created a very large map of Russia which contains about 10 screenfuls of image. The screen becomes a window to the map. The user can scroll about the entire map with a joystick. The system is very memory efficient; the entire map program plus data plus display list and character set definitions requires a total of about 4K of RAM.

There are many other applications of this technique. Any very large image that can be drawn with character graphics is amenable to this system. (Scrolling does not require character graphics. Map graphics are less desirable for scrolling applications because of their large memory requirements.) Large electronic schematics could be presented in this way. The joystick could be used both to scroll around the schematic and to indicate particular components that the user wishes to address. Large blueprints or architectural diagrams could also be displayed with this technique. Any big image that need not be seen in Its entirety can be presented with this system.

Large blocks of text are also usable here, although it might not be practical to read continuous blocks of text by scrolling the image. This system is more suited to presenting blocks of independent text. One particularly `exciting idea is to apply this system to menus. The program starts by presenting a welcome sign on the screen with signs indicating submenus pointing to other regions of the larger image. "This way to addition" could point up while "this way to subtraction" might point down. The user scrolls around the menu with the joystick, perusing his options. When he wishes to make a choice, he places a cursor on the option and presses the red button. Although this system could not be applied to all programs, it could be of great value to certain types of programs.

There are two "blue sky" applications of fine scrolling which have not yet been fully explored. The first is selective fine scrolling, in which different mode lines of the display have different scroll bits enabled. Normally you would want the entire screen to scroll, but it is not necessary, to do so. You could select one line for horizontal scrolling only, another line for vertical scrolling only, and so forth. The second blue sky feature is the prospect of using display list interrupts to change the HSCROL or VSCROL registers on the fly. However, changing VSCROL on the fly is a tricky operation; it would probably confuse ANTIC and produce undesirable results. Changing HSCROL is also tricky but might be easier.

Chapter VII

Sound

7 Sound

SOUND
The ATARI 400/800TM Home Computers have extensive hardware sound capabilities. There are four independently controllable sound channels, all able to play simultaneously. Each channel has a frequency register determining the note, and a control register regulating the volume and the noise content. Several options allow you to insert high- pass filters, choose clock bases, set alternate modes of operation, and modify polynomial counters.
DEFINITION OF TERMS AND CONVENTIONS
For the purposes of this discussion, a few terms and conventions need to be clarified:

1 Hz (Hertz) is 1 pulse per second
1 KHz (kilo-Hertz) is 1,000 pulses per second
1 MHz (mega-Hertz) is 1,000,000 pulses per second

A "pulse" is a sudden voltage rise followed somewhat later by a sudden voltage drop. If a pulse is sent to the television speaker, it will be heard as a single pop.

A "wave" as used here is a continuous series of pulses. There are different types of waves, distinguished by the shape of the individual pulses. Waves created by the ATARI Computer are square waves (as in Figure 7-2). Brass instruments typically produce triangle waves, and a singer produces sine waves (depicted in Figure 7-15).

A shift register is like a memory location (in that it holds binary data) that, when so instructed, shifts all its bits to the right one position; i.e., bit 5 will get whatever was in bit 4, bit 4 will get whatever was In bit 3, etc. Thus, the rightmost bit is pushed out, and the leftmost bit assumes the value on its input wire (see Figure 7-1).

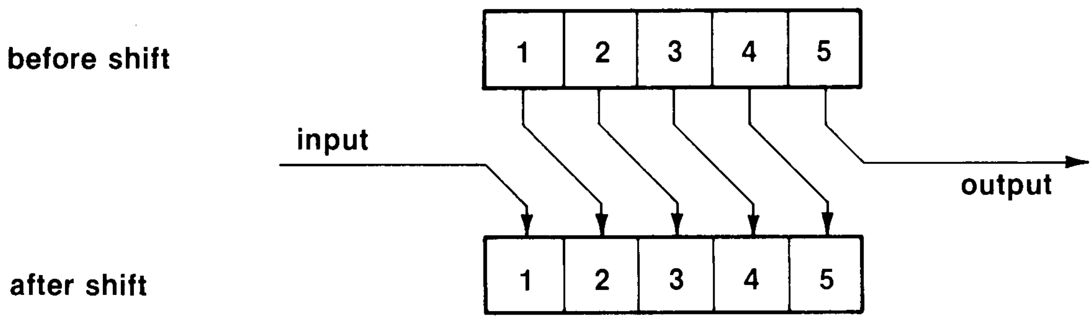


Figure 7-1 Diagram of Bit Flow of a Shift Register

AUDF1-4 is to be read, "any of the audio frequency registers, 1 through 4." Their addresses are: $D200, $D202, $D204, $D206 (53760, 53762, 53764, 53766).

AUDC1-4 is to be read, "any of the audio control registers, 1 through 4." Their addresses are: $D201, $D203, $D205, $D207 (53761, 53763, 53767).

For the purposes of this discussion, frequency is a measure of the number of pulses in a given amount of time; 1 e., a note with a frequency of 100 Hz means that in one second, exactly 100 pulses will occur. The more frequent (hence the term. "frequency") the pulses of a note, the higher the note. For example, a singer sings at a high frequency (perhaps 5 KHz), and a cow moos at a low frequency (perhaps 100 Hz). The words "frequency," "note," "tone," and "pitch" are used interchangeably.

"Noise" and "distortion" are used interchangeably although their meanings are not the same. "Noise" is a more accurate description of the function performed by the ATARI Computer.

The 60-Hz interrupt referred to later in this section is also called the vertical blank interrupt.

All examples are in BASIC unless otherwise stated. Type the examples exactly as they appear. If there are no line numbers, don't use any; and if several statements are on the same line, type them as such.

SOUND HARDWARE
Sound is generated in the ATARI computer by the POKEY chip, which also handles the serial 1/0 bus and the keyboard. The POKEY chip must be initialized before it will work properly. Initialization is required after any serial bus operation (cassette, disk drive, printer, or RS-232 read/ write). To initialize POKEY in BASIC, execute a null sound statement; i.e., SOUND 0,0,0,0. In machine language, store a 0 at AUDCTL ($D208 = 53768), and a 3 at SKCTL ($D20F = 53775, shadowed at $232 = 562).

AUDF1-4
Each audio channel has a corresponding frequency register that controls the note played by the computer. The frequency register contains the number `IN" used in a divide-by-N circuit. This divide is not a division in the mathematical sense, but rather something much simpler: for every N pulses coming in, 1 pulse goes out. For example, Figure 7-2 shows a divide-by-4 function:

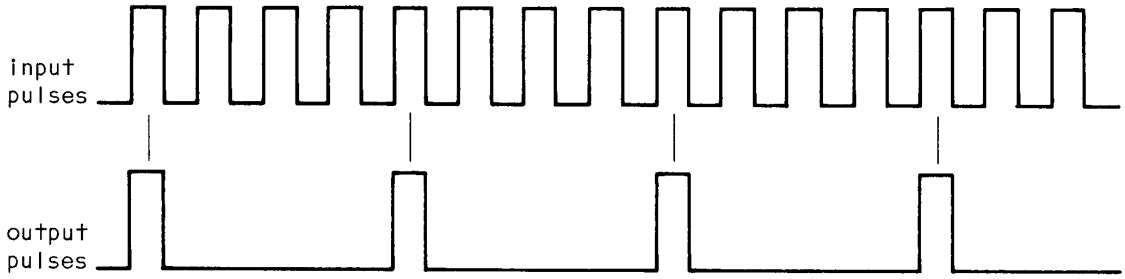


Figure 7-2 Divide-by-4 Operation

As N gets larger, output pulses will become less frequent, making a lower frequency note.

AUDC1-4
Each channel also has a corresponding control register. These registers allow the volume and distortion content of each channel to be set. The bit assignment for AUDC1-4 is as follows:

```
AUDC1-4
Bit number        1       2       3       4       5       6       7       8
               ---distortion---  --vol--  ---------volume-------
                                 --only-
```

Figure 7-3 AUDC1-4 Bit Assignment

Volume
The volume control for each audio channel is straightforward. The lower 4 bits of the audio control register contain a 4-bit number that specifies the volume of the sound. A zero in these bits means zero volume, and a 15 means as loud as possible. The sum of the volumes of the four channels should not exceed 32, since this forces overmodulation of the audio output. The sound produced tends to actually lose volume and assume a buzzing quality.

Distortion
Figure 7-3 shows that each channel also has three distortion control bits in its audio control register. Distortion is used to create special sound effects any time a pure tone is undesirable.

The computer's use of distortion offers great versatility and controllability. It is easy to synthesize of an almost endless variety of sounds, from rumbles, rattles, and squawks to clicks, whispers, and mood setting background tempos.

Distortion as used here is not equivalent to the standard interpretation. For example, "intermodulation distortion" and "harmonic distortion" are quality criteria specified for high-fidelity

stereo systems. These types of distortion refer to waveform degeneration, where the shape of the wave is slightly changed due to error in the electronic circuitry. The computer's distortion does not alter waves (they are always square waves), but rather deletes selected pulses from the waveform. This technique is not adequately characterized by the word "distortion." A more descriptive and appropriate term for these distortion methods is. "noise".

Before you can fully grasp what we mean by distortion, you must understand polynomial counters (poly-counters). Poly counters are employed in the ATARI Computer as a source of random pulses used in noise generation. The ATARI Computer's poly-counters utilize a shift register working at 1.79 MHz. The shift register's contents are shuffled and fed back into the input; this produces a semi-random sequence of bits at the output of the shift register.

For example, in the diagram below, the old value of bit 5 will be pushed out of the shift register to become the next output pulse, and bit 1 will become a function of bits 3 and 5:

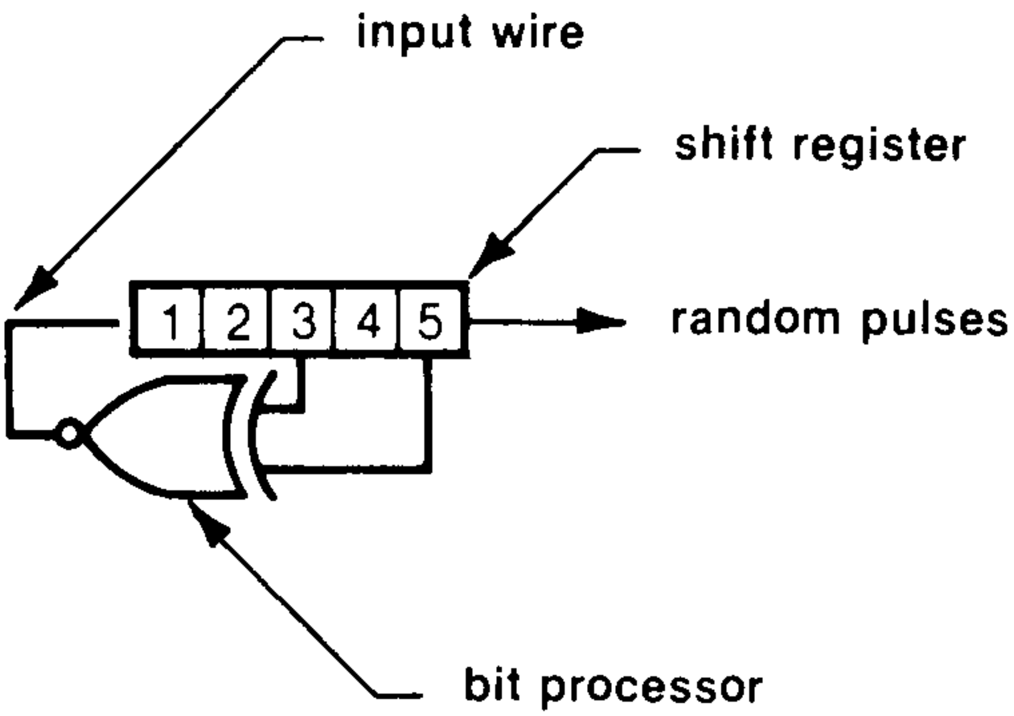


Figure 7-4 5-Bit Poly-Counter

The bit processor gets values from certain bits in the shift register (bits 3 and 5 above), and processes them in a way irrelevant to this discussion. It yields a value that becomes bit 1 of the poly-counter's shift register.

These poly-counters are not truly random because they repeat their bit sequence after a certain span of time. As you might suspect, their repetition rate depends upon the number of bits in the poly-counter; i.e., the longer ones require many cycles before they repeat, while the shorter ones repeat more often.

On the ATARI Computer, distortion is achieved by using random pulses from these poly-counters in a selection circuit. This circuit is actually a digital comparator, but "selection circuit" is more descriptive. The only pulses making it through the selection circuit to the output are those coinciding with a random pulse. Various pulses from the input are thereby eliminated in a random fashion. Figure 7-5 illustrates this selection method. A dotted line connects pulses that coincide.

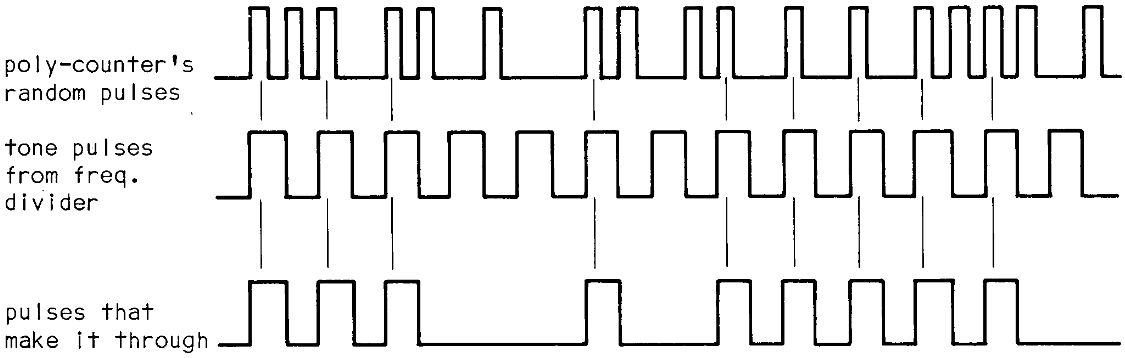


Figure 7-5 Selection Function Used To Mix In Distortion

The net effect is this: some pulses from the frequency divider circuit are deleted. Obviously, If some of the pulses are deleted, the note will sound different. This Is how distortion is introduced into a sound channel.

Because poly-counters repeat their bit sequences, their output pattern of pulses is cyclic. And since the selection circuit uses this output pattern to delete pulses from the original note, the distorted note will contain the same repetitious pattern. This allows the hardware to create noises such as drones, motors, and other sounds having repetitive patterns.

The ATARI Computer is equipped with three poly-counters of different lengths, which can be combined in many ways to produce interesting sound effects. The smaller poly-counters (4 and 5 bits long) repeat often enough to create droning sounds that rise and fall quickly; while the larger poly- counter (17 bits long) takes so long to repeat that no pattern to the distortion can be readily discerned. This 17-bit poly-counter can be used to generate explosions, steam, and any sound where random crackling and popping is desired. It is even Irregular enough to be used to generate white noise (an audio term meaning a hissing sound).

Each audio channel offers six distinct combinations of the three poly- counters:

Notes:
"Clock" means the input frequency
An "X" means, "it doesn't matter if this bit is set or not."

AUDC1-4

7	6	5	4	3	2	1	0

7	6	5	
0	0	0	div clock by freq, select using 5 bit then 17 bit polys, div by 2
0	X	1	div clock by freq, select using 5-bit poly, then div by 2
0	1	0	div clock by freq, select using 5-bit then 4 bit polys, div by 2
1	0	0	div clock by freq, select using 17-bit poly, div by 2
1	X	1	div clock by freq, then div by 2 (no poly-counters)
1	1	0	div clock by freq, select using 4 bit poly, div by 2

Figure 7-6 Available Poly-Counter Combinations

These upper AUDC1-4 bits control three switches in the audio circuit as shown below. This diagram will help you understand why the table of Figure 7-6 is structured as it is:

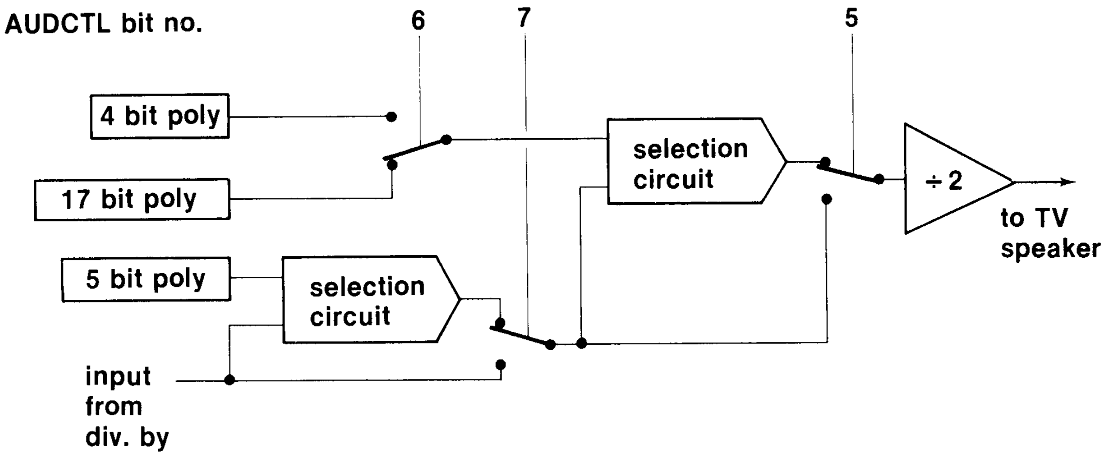


Figure 7-7 AUDC1-4 Block Diagram

Each combination of the poly-counters offers a unique sound. Furthermore, the distorted sounds can sound quite different at different frequencies. For this reason some trial and error is necessary to find a combination of distortion and frequency that produces the desired sound effect. Below is a table of guides, just to get you started:

AUDC1-4

7 6 5 4 3 2 1 0

7 6 5	low frequencies — middle frequencies — high frequencies
0 0 0	geiger counter, raging fire, rushing air, steam
0 X 1	machine gun, auto at idle, electric motor, power transformer
0 1 0	calm fire, laboring auto, auto with a "miss"
1 0 0	building crashing in, radio interference, waterfall
1 X 1	pure tones
1 1 0	airplane, lawn mower, electric razor

Figure 7-8 Sounds Produced by Distortion Combinations at Several Frequencies

Volume Only Sound
Bit 4 of AUDC1-4 specifies the volume only mode. When this bit is set, the volume value in AUDC1-4 bits 0-3 is sent directly to the television speaker; it is not modulated with the frequency specified in the AUDF1- 4 registers.

To fully understand the use of this mode of operation, you must understand how a speaker works and what happens to the television speaker when it receives a pulse. Any speaker has a cone that moves in and out. The cone's position at any time is directly proportional to the voltage it is receiving from the computer at that time. If the voltage sent is zero, then the speaker is in the resting position. Whenever the cone changes position, it moves air that is detected by your ear as sound.

From our definition of a pulse, you know that it consists of a rising voltage followed by a falling voltage. If you were to send the speaker a pulse, it would push out with the rising voltage and pull back with the falling voltage, resulting in a wave of air that can be detected by your ear as a pop. The following statements will produce such a pop on the television speaker by sending a single pulse:

```
POKE 53761,31:POKE 53761,16
```

A stream of pulses (or wave) would set the speaker into constant motion, and a continuous buzz or

note would be heard. The faster the pulses are sent, the higher the note. This is how the computer generates sound on the television speaker.

It is essential to note that in the volume only mode the volume sent does not drop back to zero automatically, but rather remains constant until the program changes it. The program should modulate the volume often enough to create a noise. Now try the following statements, listening carefully after each:

```
POKE 53761,31
POKE 53761,31
```

The first time you heard a pop, which is as expected. The speaker pushed out and moved air. But the second time you didn't. This is because the speaker cone was already In the extended position; another extension command did nothing to the speaker, moving no air, so you heard nothing. Now try this:

```
POKE 53761,16
POKE 53761,16
```

Just as before, you heard a pop the first time as the speaker moved back to its resting position, and you heard nothing the second time because the speaker was already in the resting position.

Thus, the volume only bit gives the program complete control over the position of the speaker at any time. Although the examples given above are only binary examples (either on or off), you are by no means limited to this type of speaker modulation. You may set the speaker to any of 16 distinct positions.

For example, a simple triangle wave (similar to the waveform produced by brass instruments) could be generated by sending a volume of 8 followed by 9, 10, 11, 10, 9, 8, 7, 6, 5, 6, 7, and back to 8, and repeating this sequence over and over very rapidly. By changing the volume quickly enough, virtually any waveform can be created. It is feasible, for example, to perform voice synthesis using this technique. It requires the use of assembly language. There is more discussion of this bit in a later section.

AUDCTL

In addition to the independent channel control bytes (AUDC1-4), there is an option byte (AUDCTL) affecting all four channels. Each bit in AUDCTL is assigned a specific function:

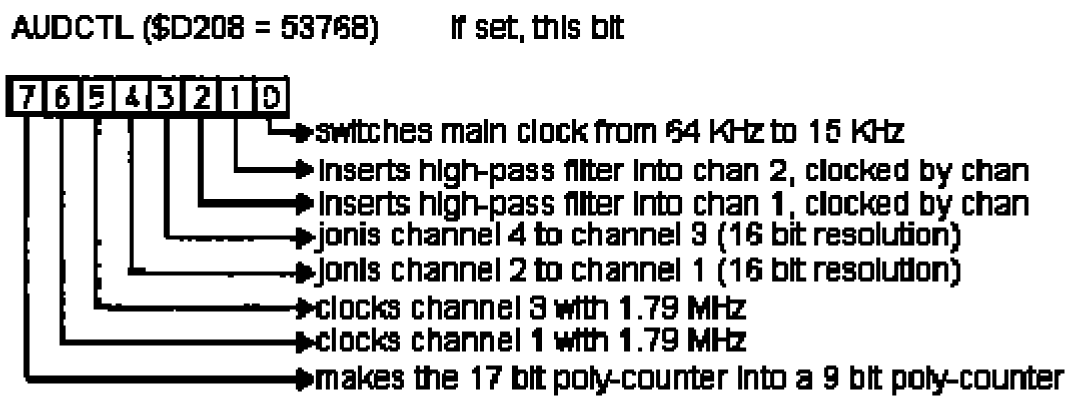


Figure 7-9 AUDCTL Bit Assignment

Clocking

Before proceeding with the explanations of the AUDCTL options, a new concept must be explained: clocking. In general, a clock is a train of pulses used to synchronize the millions of internal operations occurring every second in any computer. The central clock pulses continuously, each pulse telling the circuitry to perform another step in its operations. You may remember that a divide-by-N frequency divider outputs one pulse for every Nth input pulse. You may have wondered where the input pulses come from. There is one main input clock running at 1.79 MHZ; it can provide the input pulses. There are also several secondary clocks that can be used as input

clocks. The AUDCTL register allows you to select which clock is used as the input to the divide-by-N circuit. If you select a different input clock, the output from the frequency divider will change drastically.

For example, imagine that you are using the 15 KHz clock, and the frequency register is set to divide by 8. The rate of output pulses from the divide-by-N circuit would be about 2 KHz. But if you changed the selection of clocks to get the 64 KHz clock and did not change the frequency register, then what would happen? The divide-by-N would still be putting out one pulse for every 8th input pulse, but the input rate would be 64 KHz. The result is an output frequency (from the divide- by-N) of 8 KHz.

The formula for the output frequency (from the divide-by-N) is quite simple:

clock output frequency = clock / N

Setting bit 1 of the AUDCTL register switches from the 64-KHz clock to the 15 KHz clock. It is important to note that if this bit is set, every sound channel clocked with the 64 KHz clock will instead use the 15 KHz clock. Similarly, by setting bits 5 or 6, you can clock channels 3 or 1, respectively, with 1.79 MHz. This will produce a much higher note, as demonstrated with the following example:

```
SOUND 0,255,10,8      Turn on channel 1, low tone
POKE 53768,64         Set AUDCTL bit 6
```

16-Bit Frequency Options
The eight bits of resolution in the frequency control registers seems to provide more than adequate resolution for the task of selecting any desired frequency. There are, however, situations in which eight bits are inadequate. Consider for example what happens when we execute the following statements:

```
FOR 1=255 TO 0 STEP -1:SOUND 0,1,10,8:NEXT I
```

The sound initially rises smoothly, but as it approaches the end of its range the frequency takes larger and larger steps which are noticeably clumsy. This is because we are dividing the clock by smaller and smaller numbers. 15 KHz divided by 255 is almost the same as 15 KHz divided by 254; but 15 KHz divided by 2 is very far from 15 KHz divided by 1. The only way to solve this problem is to use a larger number that allows us to specify our frequency with greater precision. The means to do this is built into POKEY.

AUDCTL bits 3 and 4 allow two channels to be joined, creating a single channel with an extended dynamic frequency range. Normally, each channel's frequency divider number can range from 0 to 255 (8 bits of divide-by-N capability). Joining two channels allows a frequency range of 0 to 65535 (16 bits of divide-by-N capability). In this mode, it Is possible to reduce the output frequency to less than one Hertz. The following program uses two channels in the 16-bit mode, and two paddles as the frequency inputs. Insert a set of paddles into port 1, type in and run the following program:

```
10 SOUND 0,0,0,0                         Initialize sound
20 POKE 53768,80                         Clock ch1 w 1.79 MHz, clock ch2 w ch1
30 POKE 53761,160:POKE 53763,168         Turn off ch1, turn on ch2 (pure
tones)
40 POKE 53760,PADDLE(0):POKE 53762,PADDLE(l)
50 GOTO 40                               set paddles to put freqs in freq
regs
```

The right paddle tunes the sound coarsely, and the left paddle finely tunes the sound between the coarse increments.

This program first sets bits 4 and 6 of AUDCTL which means, "clock channel 1 with 1.79 MHz, and join channel 2 to channel 1." Once this happens, the 8-bit frequency registers of both channels are assumed to represent a single 16-bit number N, used to divide the input clock. Next, channel 1's volume is set to zero. Since channel 1 no longer has its own direct output, its volume setting is meaningless to us and we zero it. Channel 1's frequency register is used as the fine or low byte In the sound generation, and channel 2's frequency register is the coarse or high byte. For example, pokeing a 1 into channel 1's frequency register makes the pair divide by 1. Pokeing a 1 into

channel 2's frequency register makes the pair divide by 256. And pokeing a 1 into both frequency registers makes the pair divide by 257.

Bit 3 of AUDCTL can be used to join channel 4 to channel 3 in precisely the same way.

The following instructions demonstrate some interesting aspects of 16- bit sound.

```
SOUND 0,0,0,0
POKE 53768,24
POKE 53761,168
POKE 53763,168
POKE 53765,168
POKE 53767,168
POKE 53760,240:REM try pokeing other numbers into these next 4 locations
POKE 53764,252
POKE 53762,28
POKE 53766,49
```

High-Pass Filters
AUDCTL bits 1 and 2 control high-pass filters in channels 2 and 1 respectively. A high-pass filter allows only higher frequencies to pass through. In the case of these high-pass filters, high frequencies are defined to be anything higher than the output of another channel selected by the AUDCTL bit combination. For example, if channel 3 Is playing a cow's moo, and AUDCTL bit 2 is set, then only sounds with frequencies higher than the moo will be heard on channel 1 (anything lower than the "moooo" will be filtered out):

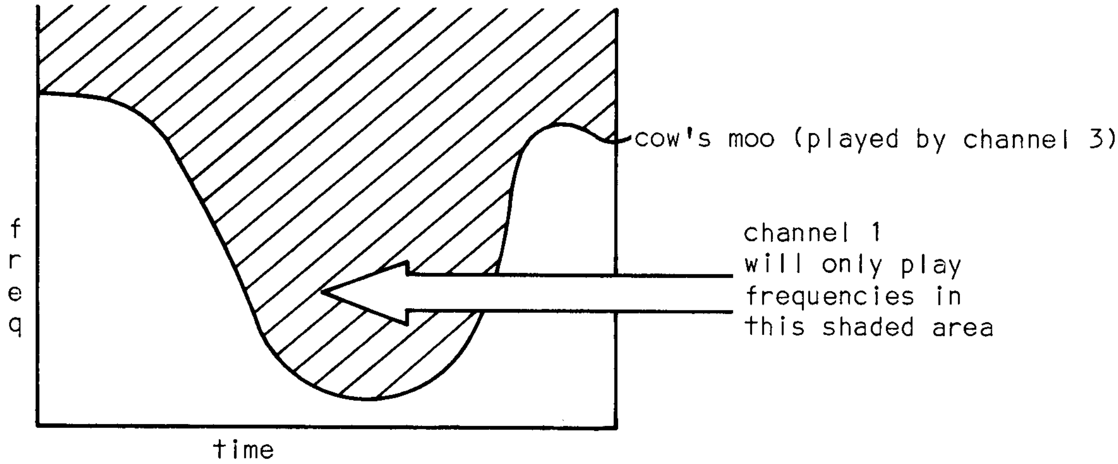


Figure 7-10 The Effect of a High-Pass Filter Inserted in Channel 1 and Clocked by Channel 3

The filter is programmable in real time since the filtering channel can be changed on the fly. This opens a large field of possibilities to the programmer. The filters are used mostly to create special effects. Try the following statements:

```
SOUND 0,0,10,0
POKE 53768,4
POKE 53761,168:POKE 53765,168
POKE 53760,254:POKE 53764,127
```

9-Bit Polynomial Conversion
Bit 7 of AUDCTL turns the 17-bit poly-counter into a 9-bit poly- counter. The shorter the poly-counter, the more often its distortion pattern repeats, or the more discernible the pattern in the distortion. Therefore, changing the 17-bit poly counter into a 9- bit poly counter will make the noise pattern more repetitious and more discernible. Try the following demonstration of the 9-bit poly counter option, listening carefully when the POKE is executed:

```
SOUND 0,80,8,8       Use the 17-bit poly
```

```
POKE 53768,128       Change to the 9-bit poly
```

SOUND GENERATION SOFTWARE TECHNIQUES

There are two basic ways to use the ATARI Computer sound system: static and dynamic. Static sound generation is the simpler of the two; the program sets a few sound generators, turns to other activities for a while, and then turns them off. Dynamic sound generation is more difficult; the computer must continuously update the sound generators during program execution. For example:

Static Sound   Dynamic Sound

```
SOUND 0,120,8,8   FOR X=0 TO 255
  SOUND 0,X,8,8
NEXT X
```

Static Sound

Static sound is normally limited to beeps, clicks, and buzzes. There are exceptions. Two examples are the programs given as special effects in the sections on high-pass filters and 16 bit sound. Another way to obtain interesting effects is to use interference, as in this example:

```
SOUND 0,255,10,8
SOUND 1,254,10,8
```

The strange effect is a result of closely phased peaks and valleys. Examine Figure 7-11. It shows two channels independently running sine waves at slightly different frequencies, and their sum. The sum curve shows the strange interference pattern created when these two channels are added.

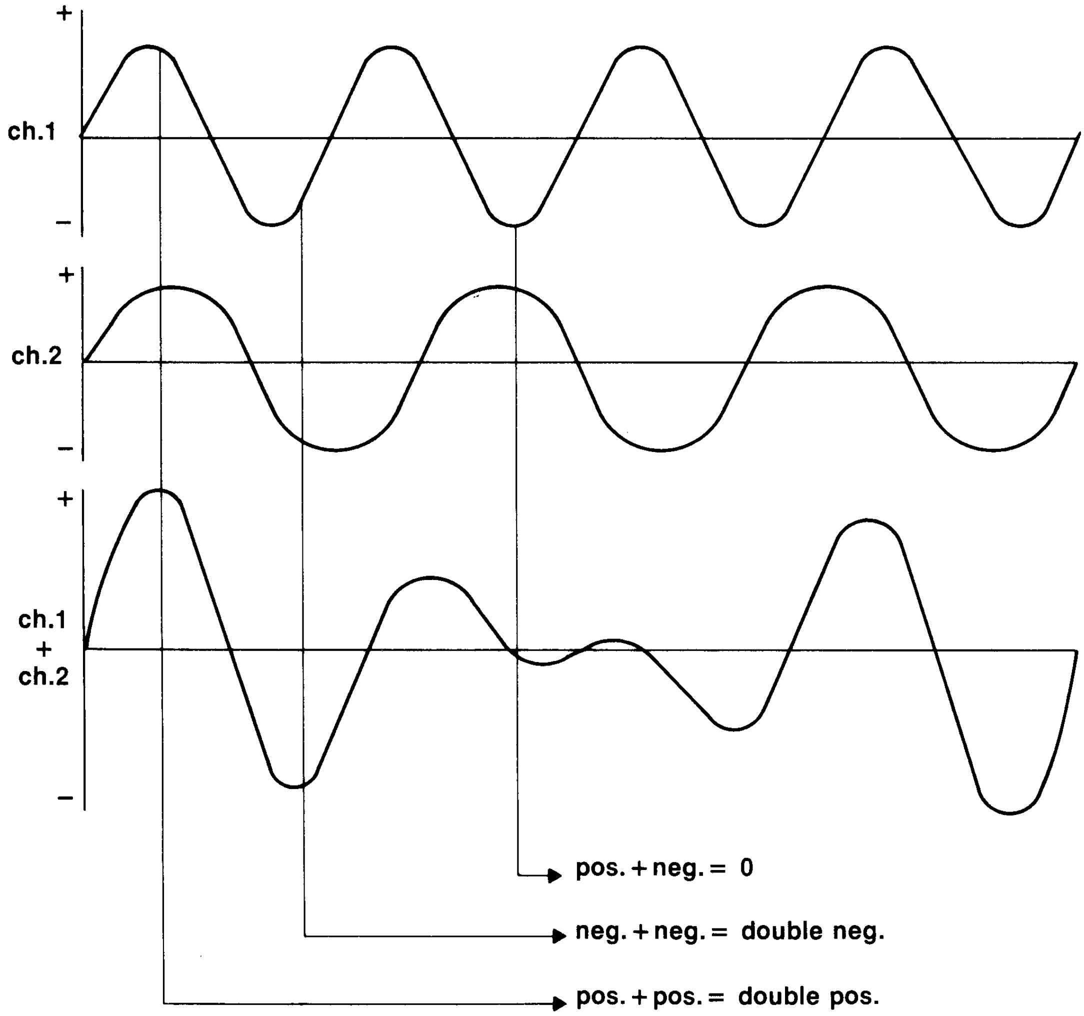


Figure 7-11 Two Sine Waves at Different Frequencies and Their Sum

Figure 7-11 shows that at some points in time the waves are assisting each other, and at other points, they interfere with each other. Adding the volumes of two waves whose peaks coincide will yield a wave with twice the strength or volume. Similarly adding the volumes of two waves while one is at maximum and the other is at minimum will result In a cancellation of both of them. On the graph of the sum curve, we can see this effect. Toward the ends of the graph, volume Increases since both channels' peaks and valleys are close together, almost doubling the sound. Toward the middle of the graph, the waves oppose each other and the resulting wave Is flat. An Interesting project might be writing a program to plot interaction patterns of 2, 3, and 4 channels as in Figure 7-11. You might discover some unique sounds.

The slighter the difference in frequency between the two channels, the longer the pattern of repetition. To understand this, draw some graphs similar to Figure 7-11 and study the Interaction. As an example, try the following statements:

```
SOUND 0,255,10,8
SOUND 1,254,10,8
SOUND 1,253,10,8
SOUND 1,252,10,8
```

As the difference in frequency grows, the period of repetition decreases.
Dynamic sound
More complex sound effects normally require the use of dynamic sound techniques. Three methods of dynamic sound generation are available to the ATARI 400/800 programmer: sound in

BASIC, 60-Hz Interrupt sound, and sound in machine code.
BASIC Sound
BASIC is somewhat limited in its handling of sound generation. As you may have noticed, the SOUND statement kills any special AUDCTL setting. This problem can be avoided by poking values directly Into the sound registers rather than using the SOUND statement.

In addition, BASIC is limited on account of its speed. If the program is not completely dedicated to sound generation, there Is seldom enough processor time to do more than static sound or choppy dynamic sound. The only alternative is to temporarily halt all other processing while generating sound.

Another problem can occur when using the computer to play music on more than one channel. If all four channels are used, the time separation between the first sound statement and the fourth can be substantial enough to make a noticeable delay between the different channels.

The following program presents a solution to this problem:

```
10 SOUND 0,0,0,0:DIM SIMUL$(16)
20 RESTORE 9999:X=1
25 READ Q:IF Q<>-1 THEN SIMUL$(X)=CHR$(Q):X=X+1:GOTO 25
27 RESTORE 100
30 READ F1,C1,F2,C2,F3,C3,F4,C4
40 IF F1=-1 THEN END
50 X=USR(ADR(SIMUL$),Fl,Cl,F2,C2,F3,C3,F4,C4)
55 FOR X=0 TO 150:NEXT X
60 GOTO 30
100 DATA 182,168,0,0,0,0,0,0
110 DATA 162,168,182,166,0,0,0,0
120 DATA 144,168,162,166,35,166,0,0
130 DATA 128,168,144,166,40,166,35,166
140 DATA 121,168,128,166,45,166,40,166
150 DATA 108,168,121,166,47,166,45,166
160 DATA 96,168,108,166,53,166,47,166
170 DATA 91,168,96,166,60,166,53,166
999 DATA -1,0,0,0,0,0,0,0
9000 REM
9010 REM
9020 REM this data contains the machine lang. program,
9030 REM and is read into SIMUL$
9999 DATA 104,133,203,162,0,104,104,157,0,210,232,228,203,208,246,96,-1
```

In this program, SIMUL$ is a tiny machine language program that pokes all four sound channels very quickly. A BASIC program using SIMUL$ can rapidly manipulate all four channels. Any program can call SIMUL$ by putting the sound register values inside the USR function in line 50 of the demonstration program. The parameters should be ordered as shown, with the control register value following the frequency register value, and repeating this ordering one to four times, once for each sound channel to be set.

As a speed consideration as well as a convenience, SIMUL$ allows you to specify sound for less than four channels; I.e., 1, 2, and 3 or 1 and 2, or just channel 1. Simply don't put the unused parameters inside the USR function.

SIMUL$ offers another distinct advantage to the BASIC programmer. As mentioned earlier, the AUDCTL register is reset upon execution of any SOUND statement in BASIC. However, using SIMUL$, no SOUND statements are executed, and thus the AUDCTL setting is retained.

There is another, but impractical, method of sound generation in BASIC. This method uses the volume-only bit of any of the four audio control registers. Type in and run the following program:

```
SOUND 0,0,0,0
10 POKE 53761,16:POKE 53761,31:GOTO 10
```

This program sets the volume-only bit in channel 1 and modulates the volume from 0 to 15 as fast as BASIC can. This program uses all of the processing time available to BASIC, yet it produces only a low buzz.
60-Hz Interrupt
This technique is probably the most versatile and practical of all methods available to the ATARI

Computer programmer.

Precisely every 60th of a second, the computer hardware automatically generates an interrupt. When this happens, the computer temporarily leaves the mainline program, (the program running on the system; i.e., BASIC, STAR RAIDERS'"). It then executes an interrupt service routine, which is a small routine designed specifically for servicing these interrupts. When the interrupt service routine finishes, it executes a special machine language instruction that restores the computer to the interrupted program. This all occurs in such a way (if done properly) that the program executing is not affected, and in fact has no idea that it ever stopped!

The interrupt service routine currently resident on the ATARI 400/800 Computer simply maintains timers, translates controller information, and performs miscellaneous other chores requiring regular attention.

Before the interrupt service routine returns to the mainline program, it can be made to execute any user routine; i.e., your sound generation routine. This is ideal for sound generation since the timing is precisely controlled, and especially since another program can be executing without paying heed to the sound generator. Even more impressive is its versatility. Because it is a machine language program, the interrupt sound program will lend itself equally well to a mainline program written in any language - BASIC, assembler, FORTH, PASCAL. In fact, the sound generator will require few, if any, modifications to work with another program or even another language.

A table-driven routine offers maximum flexibility and simplicity for such a purpose. "Table-driven" refers to a type of program that accesses data tables in memory for its information. In the case of the sound generator, the data tables would contain the frequency values and possibly the audio control register values. The routine would simply read the next entries in the data table, and put them into their respective audio registers. Using this method, notes could change as often as 60 times per second, fast enough for most applications.

Once such a program has been written and placed in memory (say, at location $600), you need to install it as a part of the 60-Hz interrupt service routine. This is accomplished by a method known as vector stealing.

Memory locations $224,$225 contain the address of a small routine called XITVBL (eXIT Vertical BLank Interrupt service routine). XITVBL is designed to be executed after all 60- Hz Interrupt processing is complete, restoring the computer to the mainline program as previously discussed.

The procedure to Install your sound routine Is as follows:

1. Place your program in memory. 2. Verify that the last Instruction executed Is a JMP $E462 ($E462 is XITVBL, so this will make the main-line program continue). 3.Load the x register with the high byte of your routine's address (a 6 in this case). 4.Load the y register with the low byte of your routine's address (a 0 in this case). 5. Load the accumulator with a 7. 6. Do a JSR $E45C (to set locations $224,$225). Steps 3-6 are all required to change the value of $224,$225 without error. The routine called Is SETVBV (SET Vertical Blank Vectors), which will simply put the address of your routine into locations $224,$225. Once installed, the system will work as follows when an Interrupt occurs:

1. The computer's Interrupt routine is executed. 2.It jumps to the program whose address is in $224,$225, which is now your routine. 3. Your routine executes. 4. Your routine then jumps to XITVBL. 5. XITVBL restores the computer and makes It resume normal operation.

If you do not wish to Implement such a program yourself, there is one available from the Atari Program Exchange. The package is called INSOMNIA (Interrupt Sound Initializer/Alterer). It allows creation and modification of sound data while you listen. It is accompanied by an interrupt sound generator that Is table driven and compatible with any language.
Machine-Code Sound Generation
Direct control of sound registers with mainline machine language opens new doors in sound generation. The technique is as follows: write a program similar to the 60-Hz interrupt routine in that it is table-driven, but now the mainline routine is dedicated to sound generation. By expending much more processor time on sound generation, you can produce higher quality sounds.

Consider, for example, the output of a typical 60 Hz music routine:

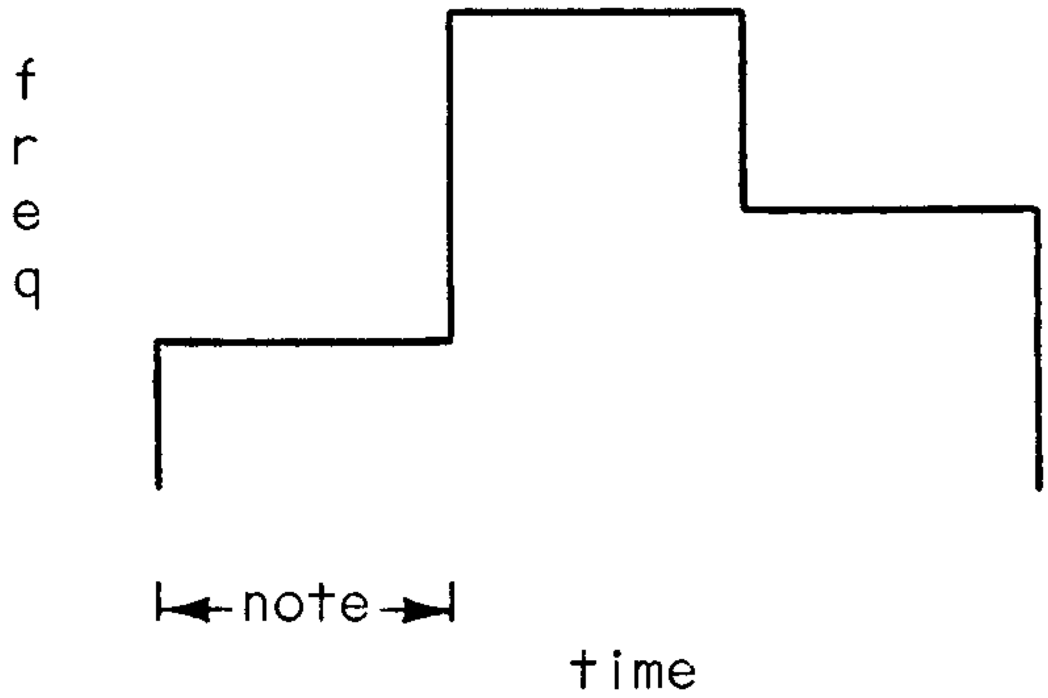


Figure 7-12 Example of 3 Music Notes Played With a 60 Hz Interrupt Music Routine

Since much more processing time is available with mainline machine language, we can change the frequency at very high speed during the note's playing time so that it simulates an instrument. For example, suppose we discovered that whenever any piano key is struck it produces a characteristic sequence of frequencies, as shown in Figure 7-13.

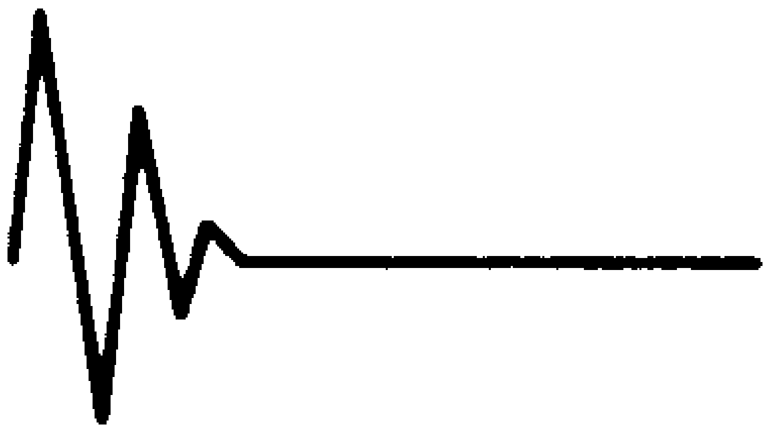


Figure 7-13 Graph of Frequency Sequence for a Piano Note

Let's call the above graph the "piano envelope". To simulate a piano, the idea would be to very quickly apply the piano envelope to the plain vanilla beep. The note is thus slightly modified during its playing time. For example, a piano simulation of the 3 notes in Figure 7-12 would look like this:

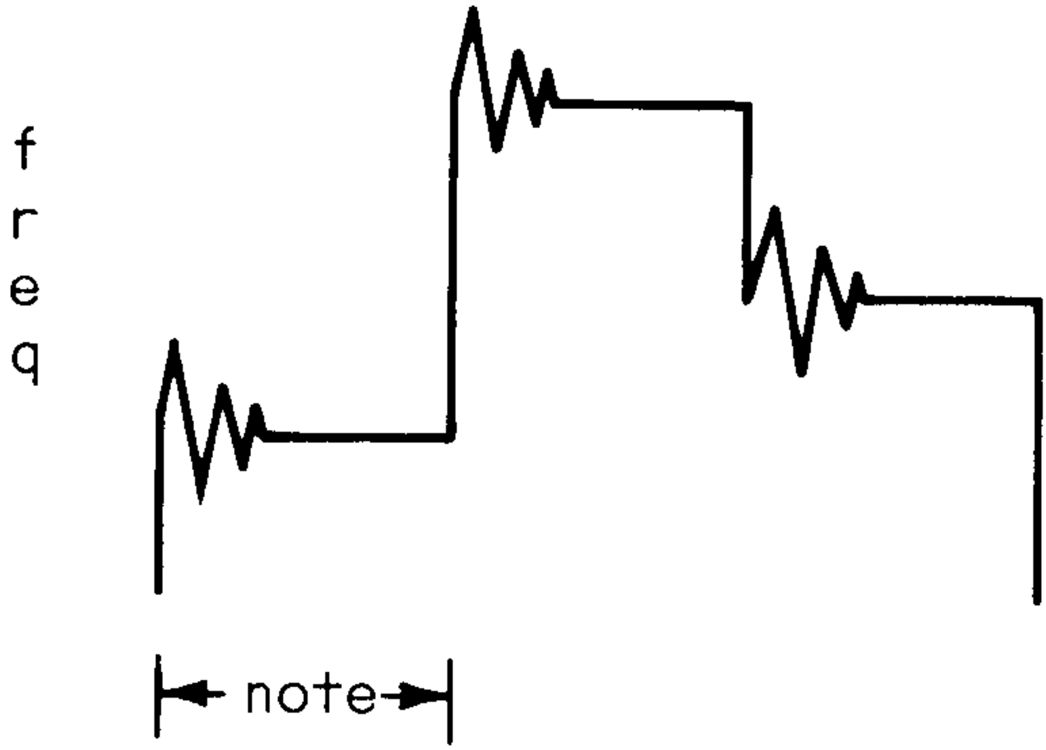


Figure 7-14 Example of the 3 Notes of Figure 7.10 Played With a Piano Envelope

We have essentially the same sound produced by the standard music routine of Figure 7-12, only the notes now have a piano tone, and sound much prettier than just the flat beeps. Unfortunately, we had to sacrifice all other processing to get that piano tone. The sound channel Is no longer updated only once every note, but perhaps 100 times within the note's duration.
Volume only sound
Earlier we experimented with the AUDC1-4 volume only bits, but discovered that they weren't of much use in BASIC. This was due entirely to the fact that BASIC Is too slow to effectively use them. This is not the case with machine language.

As mentioned earlier, this bit offers a tremendous capacity for accurate sound reproduction. True waveform generation (to the time and volume resolution limits of the computer) is made possible with this bit. instead of just putting a piano flavor into the music, you can now make it closely replicate a piano sound. Unfortunately, it can never precisely duplicate an instrument. 4 bits (16 values) is not enough volume resolution for truly high-quality work. Nevertheless, the technique does generate surprisingly good sounds. The following program demonstrates the use of one of the volume only bits. If you have an assembler, type it in and try it:

```
0100 ;
0110 ; VONLY        Bob Fraser 7-23-81
0120 ;
0130 ;
0140 ; volume-only AUDC1-4 bit test routine
0150 ;
0160 ;
0170 ;
0180 ;
```

```
D208        0190 AUDCTL =    $D208
D200        0200 AUDF1  =    $D200
D201        0210 AUDC1  =    $D201
D20F        0220 SKCTL  =    $D20F
            0230 ;
            0240 ;
0000        0250        *=   $BO
00B0 01     0260 TEMPO  .BYTE 1
00B1 00     0270 MSC    .BYTE 0
            0280
            0290
            0300
00B2        0310        *=   $4000
4000 A900   0320        LDA  #0
4002 8D08D2 0330        STA  AUDCTL
4005 A903   0340        LDA  #3
4007 8D0FD2 0350        STA  SKCTL
400A A200   0360        LDX  #0
            0370 ;
400C A900   0380        LDA  #0
400E 8D0ED4 0390        STA  $D40E     kill vbi's
4011 800ED2 0400        STA  $D20E     kill irq's
4014 8B00D4 0410        STA  $D400     kill dma
            0420
            0430
            0440
4017 BD5240 0450 L00    LDA  DTAB,X
401A 85B1   0460        STA  MSC
            0470 ;
401C BD3640 0480        LDA  VTAB,X
401F A4B0   0490 L0     LDY  TEMPO
4021 8D01D2 0500        STA  AUDC1
4024 88     0510 L1     DEY
4025 D0FD   0520        BNE  L1
            0530 ;
            0540 ; dec most sig ctr
4027 C6B1   0550        DEC  MSC
4029 D0F4   0560        BNE  L0
            0570 ;
            0580 ;
            0590 ; new note
            0600 ;
402B E8     0610        INX
402C EC3540 0620        CPX  NC
402F D0E6   0630        BNE  L00
            0640 ;
            0650 ; wrap note pointer
4031 A200   0660        LDX #0
4033 F0E2   0670        BEQ L00
            0680 ;
            0690 ;
4035 1C     0700 NC     .BYTE 28      note count
            0710 ;
            0720 ; table of volumes to be played In succession
            0730 VTAB
4036 18     0740        .BYTE 24,25,26,27,28,29,30,31
4037 19
4038 1A
4039 1B
403A IC
403B ID
403C 1E
403D IF
403E 1E     0750        .BYTE 30,29,28,27,26,25,24
403F 1D
4040 1C
4041 1B
4042 1A
4043 19
4044 18
4045 17     0760        .BYTE 23,22,21,20,19,18,17
4046 16
4047 15
4048 14
4049 13
404A 12
404B 11
404C 12     0770        .BYTE 18,19,20,21,22,23
404D 13
404E 14
```

```
404F 15
4050 16
4051 17
            0780 ;
            0790 ; this table contains the duration of each entry above
            0800 DTAB
4052 01     0810        .BYTE 1,1,1,2,2,2,3,6
4053 01
4054 01
4055 02
4056 02
4057 02
4058 03
4059 06
405A 03     0820        .BYTE 3,2,2,2,1,1,1
405B 02
405C 02
405D 02
405E 01
405F 01
4060 01
4061 01     0830        .BYTE 1,1,2,2,2,3,6
4062 01
4063 02
4064 02
4065 02
4066 03
4067 06
4068 03     0840        .BYTE 3,2,2,2,1,1
4069 02
406A 02
406B 02
406C 01
406D 01
```

Surprisingly, speed is not really a problem here. The wave has almost 60 steps, and the program can still be made to play the wave at up to 10 Khz.

Remove lines 400-410, and try the program once more. It will sound quite broken up. The cause is the 60 Hz interrupt discussed In the previous section. You can actually hear the interrupts taking pace since all sound stops during that time.

Line 420 disables screen DMA. This is why the screen goes to solid background color when the program is executed. It serves two purposes: to speed up the processor, and to make the timing consistent, since DMA steals cycles at odd intervals.

In this demonstration program, the sound created is a sine wave. The wave is remarkably pure, and does indeed sound like a sine wave. if graphed, the data looks like this:

```
15-I            ------
14-I         ---      ---
13-I       --            --
12-I     --                --
11-I   --                    --
10-I  -                        -
9-I -                          -
8=I-----------------------------------------------------------
7-I                              -                          -
6-I                               -                        -
5-I                                --                    --
4-I                                  --                --
3-I                                    --            --
2-I                                      ---      ---
1-I                                         ------
```

Figure 7-15 Graph of Sine Wave Data for Volume Only Program

This section has discussed the technical aspects of sound generation with the ATARI Computer. The programmer must also understand the broader role of sound in the complete software package.

Movie makers have long understood the importance of mood setting background music. The recent space adventure movies by George Lucas are excellent examples. When the villain enters

the room you know immediately to fear and hate him from the menacing background rhythms accompanying his entry. You gleefully clap your hands when the hero saves the princess while gallant music plays in the background. Likewise, horror films can frighten you by merely playing eerie music, even though the action may be completely ordinary.

SPACE INVADERS (trademark of Taito America Corp) issues a personal threat to its player and victim with its echoing stomp. As the tempo increases, knuckles whiten and teeth grind. When a Zylon from STAR RAIDERS"' fires a photon torpedo you push frantically on the control to avoid impact. As it bores straight for your forehead, time slows and you hear it hissing louder and louder as it approaches. Just before impact, you duck and dislodge yourself from your armchair.

Impressionistic sounds affect our subconscious and our state of mind. This Is due possibly to the fact that sounds, if present, are continuously entering our mind whether or not we are actively listening. Visual inputs, on the other hand, require the user's attention. If we are distracted from the TV set, we cease to concentrate on the picture and the image leaves our mind. Sound therefore offers the programmer a direct path to the user's mind - bypassing his thought processes and zeroing in on his emotions.

Chapter VIII

The Operating System

8 The Operating System

INTRODUCTION
With every ATARI Home Computer System comes an ATARI 10K Operating System Cartridge. The importance of this cartridge is often overlooked. Without it, you have a lot of potential, but absolutely nothing else! This situation is not unique to the ATARI Home Computer System; It is encountered with all computers. A computer is, after all, merely a collection of hardware devices. A user must manage these resources to accomplish any task. If all programmers had to start from scratch on each program, we would have an even larger software shortage than we have today. The solution that has evolved over the years is to build in a program that manages the resources available to the system, and eases the programming burden required to control them. This program is known by various names: Operating System, Master Control Program, System Executive, System Monitor, etc. In the ATARI Home Computer System it is known as the Operating System or OS.

The first task facing the student of the Operating System, is to take an inventory of exactly what resources are available to the OS. These are:

6502 Microprocessor
RAM Memory (various amounts)
ANTIC LSI Integrated Circuit
CTIA LSI Integrated Circuit
POKEY LSI Integrated Circuit
PIA Peripheral Interface Adapter Integrated Circuit

By using these resources, the OS can interact with and control a wide variety of external hardware devices, including a television receiver/ monitor, keyboard, console speaker, console switches, joysticks, paddles, cassette recorder, disk drive, printers, RS-232 interface and modem.

The remainder of this subsection briefly lists the main elements of the OS. These elements are described in detail in following subsections.

MONITOR. The OS monitor is the system routine that is executed when the computer is turned on or the SYSTEM RESET] button is pressed. Through this routine the OS takes control of the system; It does not relinquish control unless control is taken away from it by the programmer. The Monitor sets up the memory management system, initializes the I/O subsystem, sets up system vectors and selects the execution environment after initialization is complete.

INTERRUPT PROCESSING STRUCTURE. The computer utilizes the standard interrupt processing structure of the 6502 microprocessor, with some external augmentation for enhanced flexibility. Interrupts are generated by numerous events, including keyboard keystrokes, the **BREAK** keystroke, some serial bus events, system timer timeouts, and the vertical blank interval on the television.

OS SYSTEM VECTORS. The system vectors provide a mechanism that allows users to access system routines, or customize the OS for special needs. The most frequent uses of the vectors are to call I/O system routines, set timers, and transfer control to different execution environments. System routines may be vectored to in one of two ways. ROM vectors are locations that contain JMP instructions to system routines and cannot be altered. RAM vectors are RAM locations that contain alterable addresses of system routines. The locations of both types of vectors are guaranteed to remain the same in future releases of the OS.

INPUT/OUTPUT SUBSYSTEM. The OS gives an application programmer access to the full capabilities of the computer's peripherals. The Input/Output subsystem is a set of routines that link high level I/O operations with device handlers that control the physical I/O hardware.

REAL TIME PROGRAMMING. The ATARI Home Computer is well equipped to deal with problems in the "real time domain". To facilitate this feature, the OS has two types of timers: hardware timers and system software timers. Hardware timers are countdown timers that can be used to time events with durations that range from half a microsecond to several seconds. System timers are software timers that tick at 60 Hertz, and can be used for applications as diverse as serial bus

timing and sound effect generation.

ROM CHARACTER SET. The computer is equipped with what is known as a "soft character set", i.e., It can be changed. The ROM-based character set is used to provide a standard character set at power-up.

FLOATING POINT PACKAGE. The floating point package is a set of mathematical routines that extend the arithmetic capability of the system. The routines use binary coded decimal (BCD) arithmetic to provide standard mathematical functions (+, -, *, /), exponential and logarithmic functions as well as conversion from ATASCII to BCD and BCD to ATASCII.

THE MONITOR

The OS monitor is that portion of the OS ROM that handles both the power-up and SYSTEM RESET sequences. These sequences allow the OS to gain Initial control of the computer and ensure that everything is properly initialized before releasing partial control to an application program. Both sequences are similar in function and in fact share much of the same code.

The power-up routine (also known as Coldstart) is invoked either by turning on the computer or by jumping to COLDSV ($E477), a system routine vector. Important items to remember about the power-up sequence are:

1. ALL of RAM memory is cleared except locations $0000-$000F.

2. Both a cassette and disk boot are attempted. BOOT? ($0009) is a flag that indicates the success or failure of the boots. Bit 0 = 1 for a successful cassette boot; Bit 1 = 1 for a successful disk boot.

3. COLDST ($0244) is a flag that tells the Monitor that it is in the middle of power-up. COLDST=0 means the [SYSTEM RESET] key has been pressed, whereas COLDST<>O indicates initial power-up. The COLDST flag can be used to gain a certain amount of program security. If COLDST is set to a non-zero value during program execution, then pressing [SYSTEM RESET] will initiate the power-up sequence. This will prevent the user from gaining control of the computer while the program is running.

Pressing the [SYSTEM RESET] key causes a SYSTEM RESET (also known as Warmstart). Some of the key facts to remember about the SYSTEM RESET sequence are:

1. The OS RAM vectors are downloaded from ROM during both SYSTEM RESET and power-up sequences. If you wish to "steal" a vector, some provision must be made to handle SYSTEM RESET. See the MEMORY MANAGEMENT subsection of this section for suggestions on how this is done.

2. MEMLO, MEMTOP, APPMHI, RAMSIZ and RAMTOP are reset during System Reset. If you wish to alter these RAM pointers to reserve some space for assembler modules called by BASIC, you must make some provision for handling SYSTEM RESET. Figure 8-3 provides an example of how to do this.

The next few pages present a detailed flowchart for the power-up and SYSTEM RESET sequences.

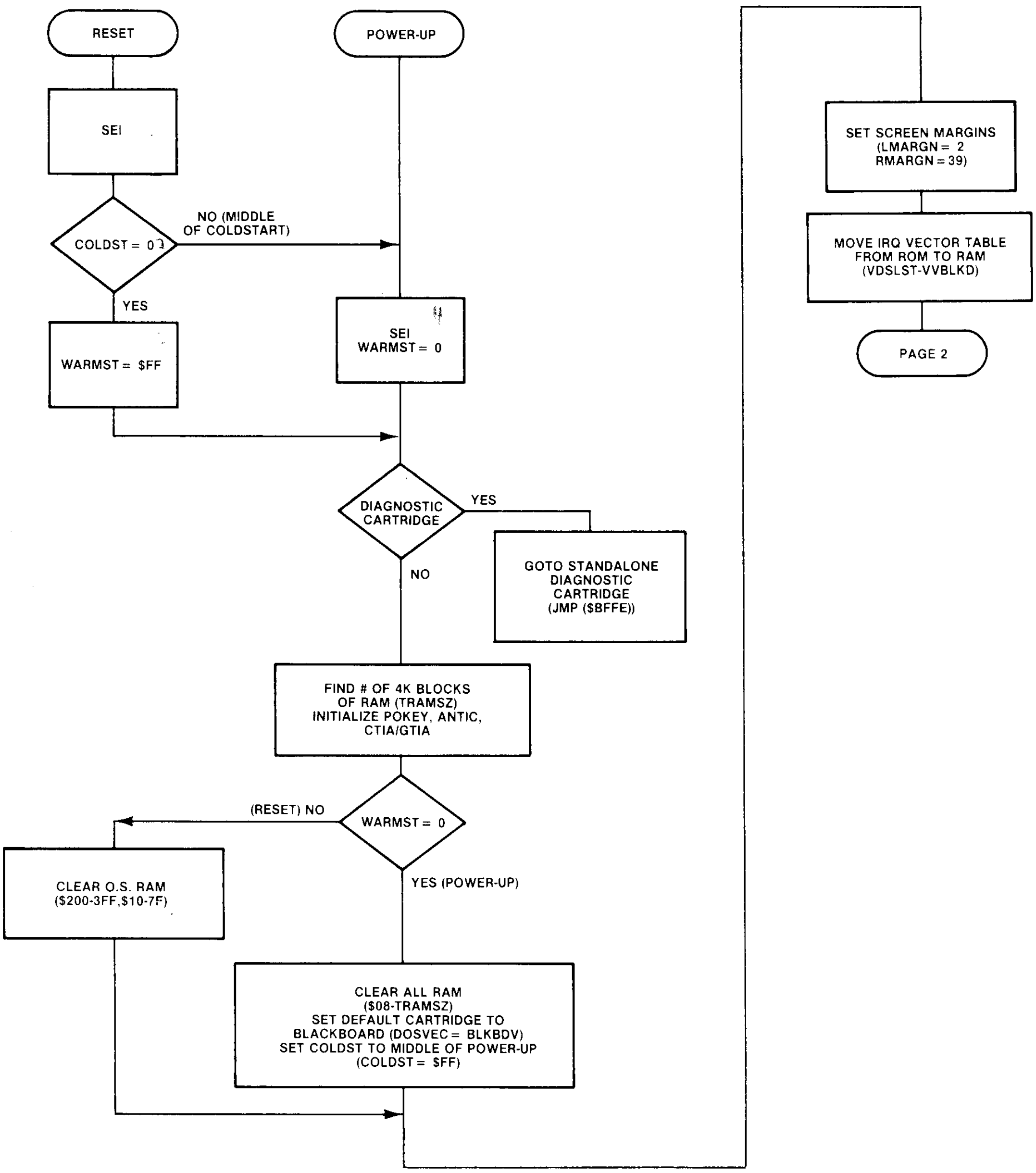
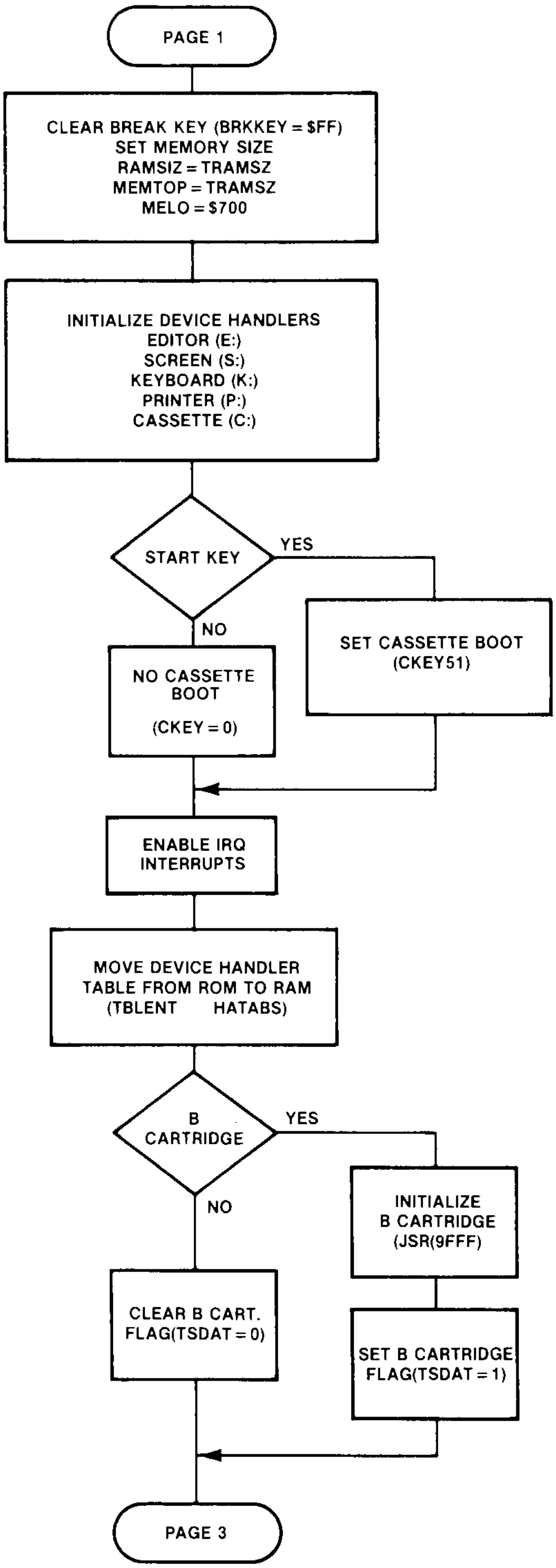


Figure 8-1.1 System Initialization



PAGE 1
CLEAR BREAK KEY (BRKKEY = $FF)
SET MEMORY SIZE
RAMSIZ = TRAMSZ
MEMTOP = TRAMSZ
MELO = $700
INITIALIZE DEVICE HANDLERS
EDITOR (E:)
SCREEN (S:)
KEYBOARD (K:)
PRINTER (P:)
CASSETTE (C:)
START KEY
YES
NO
SET CASSETTE BOOT
(CKEY51)
NO CASSETTE
BOOT
(CKEY = 0)
ENABLE IRQ
INTERRUPTS
MOVE DEVICE HANDLER
TABLE FROM ROM TO RAM
(TBLENT HATABS)
B
CARTRIDGE
YES
NO
INITIALIZE
B CARTRIDGE
(JSR(9FFF)
CLEAR B CART.
FLAG(TSDAT = 0)
SET B CARTRIDGE
FLAG(TSDAT = 1)
PAGE 3

Figure 8-1.2 System Initialization

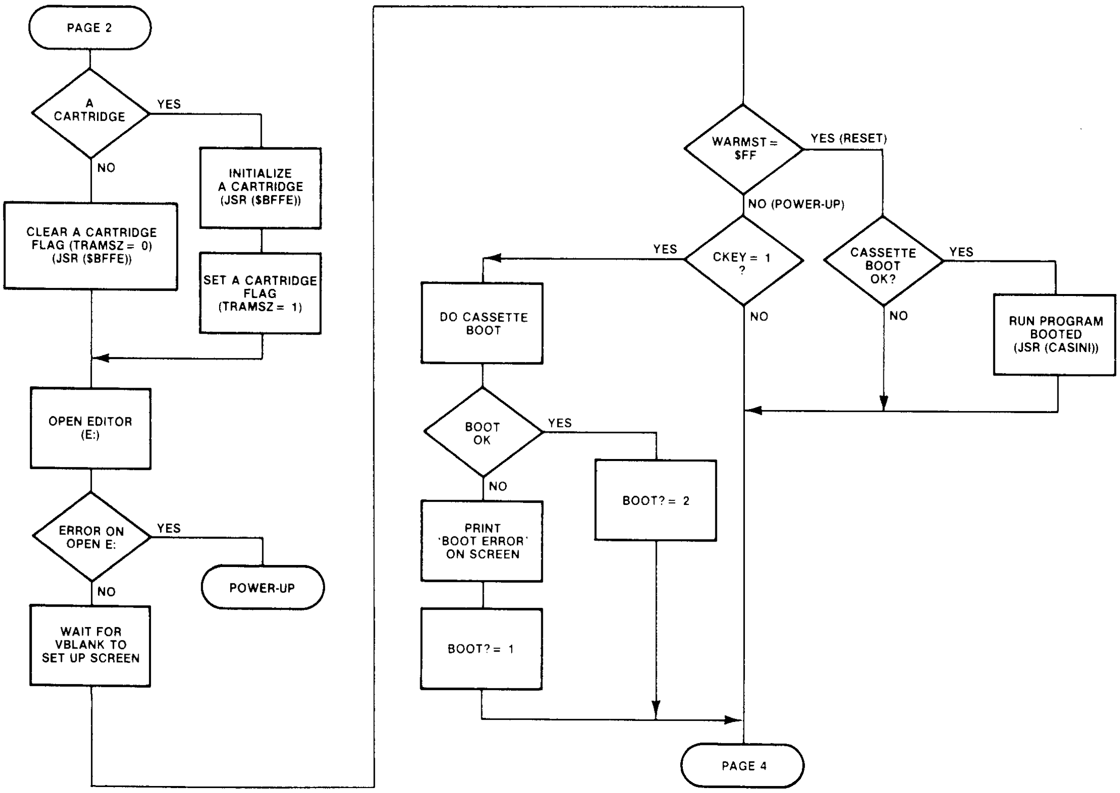


Figure 8-1.3 System Initialization

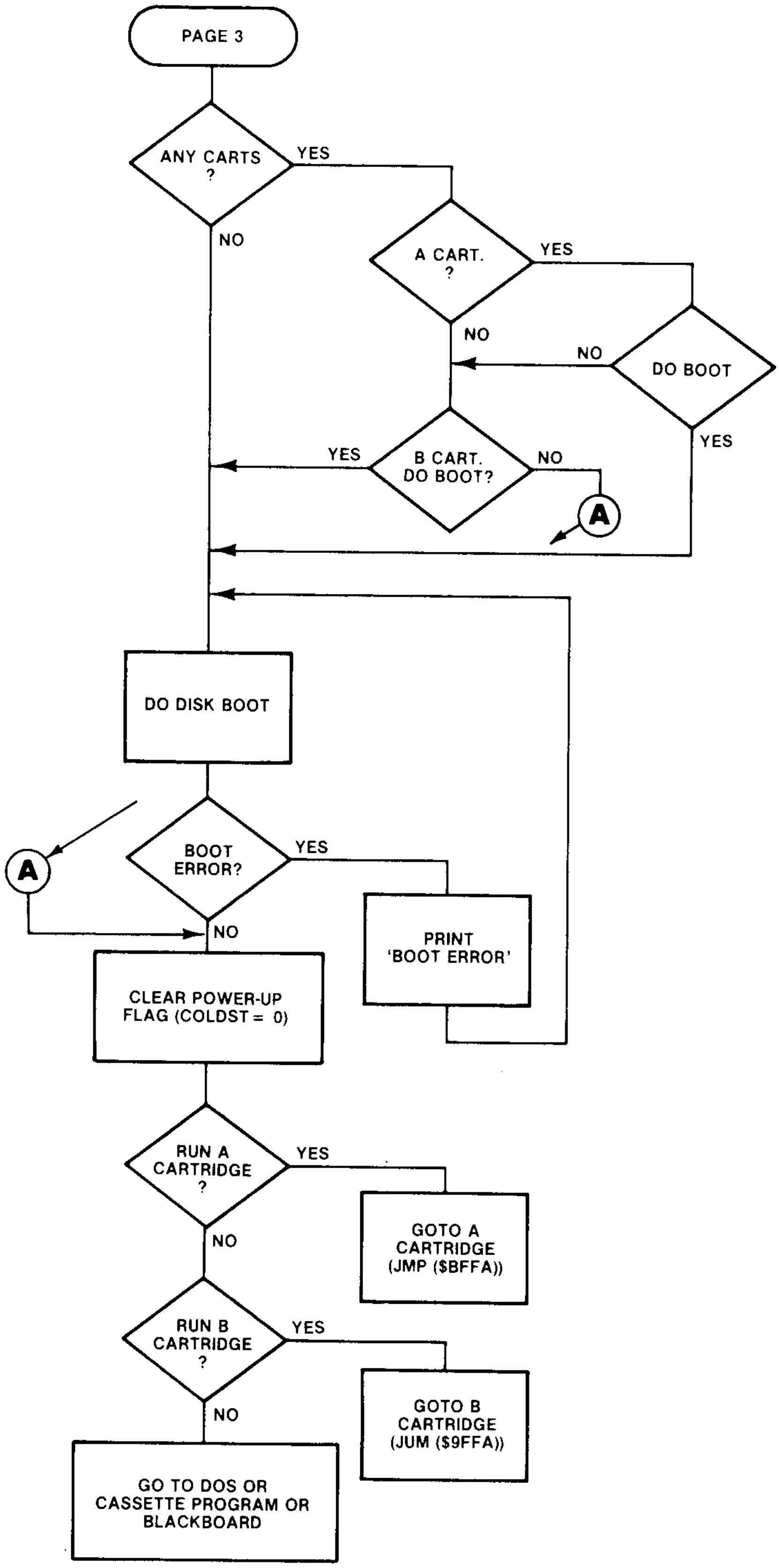


Figure 8-1.4 System Initialization

MEMORY MANAGEMENT
The fact that the OS is written for a 6502 microprocessor dictates a number of overall memory management decisions. In the 6502, there are three special regions in the memory address space. Page zero has crucial significance in that the use of data values on this page will result in tighter, faster executing code. Indeed, there are instructions that absolutely require page zero locations to work. Page one is special because it is used for the 6502 stack. Addresses $FFFA - $FFFF are also special because they are reserved for hardware reset and interrupt vectors.

Thus, the first task of memory management is to assign the OS ROM to the highest part of memory address space. The OS resides in the address space from $D800 to $FFFF. Just under this area is the space reserved for the hardware registers in ANTIC, CTIA, and POKEY. These reside in the $D000-$DFFF range.

At the other end of memory address space, the OS reserves half of page zero for its own use. Pages two, three, four and five are also reserved for OS usage. From a programming viewpoint, the usable memory area runs from $0600 to $BFFF.

When the system is powered-up, one of the first actions taken by the OS is to determine how much RAM memory is present. This is accomplished by checking the first byte of each 4K block of memory starting at $1000. The contents of this byte are read, complemented and an attempt is made to store the complemented value back out. If this attempt is successful, the temporary memory size counter is incremented by 4K. This process continues until a location is found that cannot be changed. Two variables, RAMTOP and RAMSIZ contain the number of RAM pages present. In addition to these locations, pointers MEMLO, MEMTOP, and APPMHI are maintained by the OS memory management routines. The relationships between these pointers are shown in Figure 8-2, a simple memory map.

MEMLO is a 2-byte location that the OS uses to indicate where an application program may begin. You can modify MEMLO to create reserved areas for assembly language routines that may be called from BASIC. BASIC uses the value In MEMLO to determine the starting location of a program (see Section 10 for a discussion of the structure of a BASIC program). If the value of MEMLO is changed to a higher address, it must be done before control is transferred to the BASIC cartridge. This is a tricky operation, because MEMLO is reset by both power-up and SYSTEM RESET.

If an application program is running in a disk drive environment, the AUTORUN.SYS facility can be used to change MEMLO to reserve space. However, DOS is also initialized during SYSTEM RESET via the DOSINI vector ($000C). This vector contains the address of the DOS initialization code called as part of the monitor system initialization. DOSINI is also the only point at which you can "trap" the SYSTEM RESET sequence. Since the DOS initialization must occur regardless of what is done to the MEMLO pointer, you must allow the normal initialization to occur before "stealing" the DOSINI vector. This may be done by moving the contents of DOSINI into the 2-byte address of a JSR information on the AUTORUN.SYS feature). Just after the JSR instruction, place the code which sets MEMLO to a new value. Follow this with a RTS instruction. DOSINI must then be reset to the address of the JSR instruction. When a SYSTEM RESET occurs, the new code sequence is called and the first instruction, JSR OLDDOSINI, initializes DOS. The remaining code is then executed which sets MEMLO to its new value and then rejoins the rest of the initialization sequence. Figure 8-3 presents an example showing how to do this.

The above technique can also be used with MEMTOP, the user high memory pointer. This pointer indicates the highest RAM address accessible to an application program. This RAM address differs from the highest physical RAM address because the OS allocates some RAM at the very top of RAM for its display list and display data. Space for assembly language modules and data can be set aside by lowering MEMTOP from the values set by power-up and SYSTEM RESET. Using MEMTOP instead of MEMLO to reserve space does create one problem. The value of MEMTOP depends on both the amount of RAM in the system and the graphics mode of the display. This makes it difficult to predict its value before actually examining the location unless you make assumptions about the system configuration. This uncertainty over the final location of the machine code forces the programmer to use only relocatable code.

APPMHI is a location that contains an address specifying the lowest address to which the display

RAM may extend. Correctly setting APPMHI ensures that the display handler will not clobber some of your program code or data.

RAMSIZ, like MEMTOP, can also be used to reserve space for user routines or data. Since RAMSIZ is a single byte value that contains the number of RAM pages present (i.e., groups of 256 bytes), lowering its value by 1 will reserve 256 locations. The advantage of using RAMSIZ instead of MEMTOP is that the space saved by moving RAMSIZ down is above the display memory, whereas space saved by moving MEMTOP down remains below the display memory.

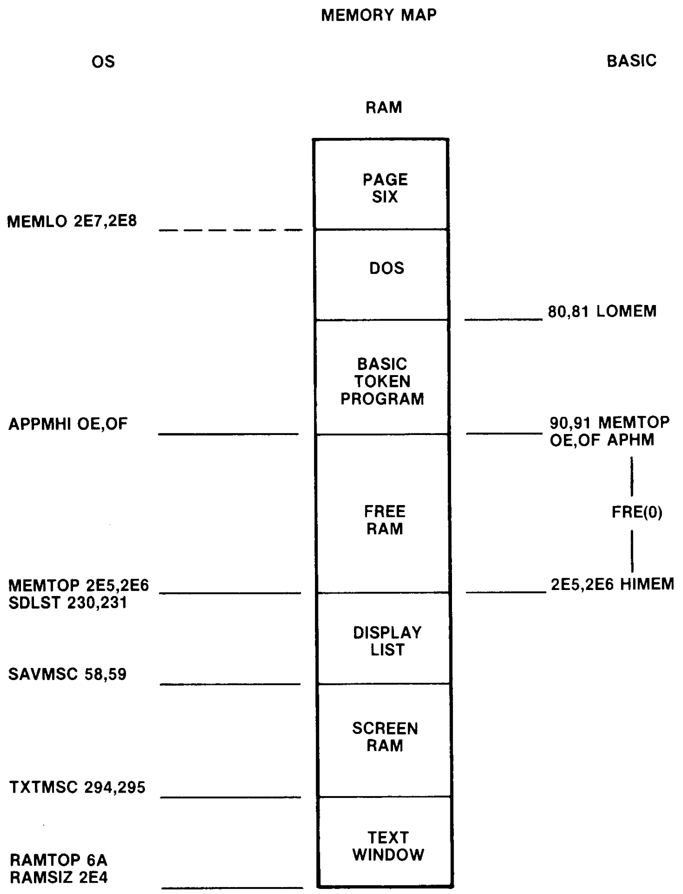


Figure 8-2 OS and BASIC Pointers (DOS present)

```
            0010 ; RESET THE MEMLO POINTER
            0020 ;
0600        0030 START    =    $600
000C        0040 DOSINI   =    $0C
02E7        0050 MEMLO    =    $2E7
3000        0060 NEWMEM   =    $3000     ; THIS IS THE NEW VALUE FOR MEMLO
            0065 ;
            0070 ; THIS ROUTINE RESERVES SPACE FOR ASSEMBLY ROUTINES
            0090 ; BY RESETING THE MEMLO POINTER. IT RUNS AS
            0100 ; AN AUTORUN.SYS FILE. IT ALSO RESETS MEMLO ON [RESET].
            0120 ; MEMLO IS SET TO THE VALUE OF NEWMEM.
            0130 ;
            0140 ; THIS PART IS PERMANENT, IE. NEEDS TO BE RESIDENT.
            0150 ; THE SYSTEM DOSINI VECTOR HAS BEEN STOLEN
            0160 ; AND STORED IN THE ADDRESS PORTION OF THE JSR TROJAN
            0170 ; INSTRUCTION. SO WHEN [RESET] IS PRESSED, DOSINI VECTORS
            0180 ; TO INITDOS, JSR TROJAN THEN CALLS THE DOS INITIALIZATION
            0185 ; ROUTINES, MEMLO IS RESET TO NEW VALUE AND  CONTROL
            0190 ; RETURNS TO THE MONITOR.
0000        0200        *=   START
```

```
                0210 INITDOS
0600 200D06 0220        JSR  TROJAN   DO DOS INITIALIZATION
0603 A900   0230        LDA  #NEWMEM&255
0605 8DE702 0240        STA  MEMLO
0608 A930   0250        LDA  #NEWMEM/256
060A 8DE802 0260        STA  MEMLO+L
            0270 TROJAN
060D 60     0280        RTS
            0290 ; THIS PART IS EXECUTED AT POWER UP ONLY AND
            0300 ; CAN BE DELETED AFTER POWER-UP.
            0330 ; THIS ROUTINE STORES THE CONTENTS OF DOSINI INTO THE JSR
            0350 ; TROJAN INSTRUCTION. IT THEN REPLACES DOSINI WITH
            0370 ; A NEW VALUE, LOCATION INITDOS.
            0390 GRABDOSI
060E A50C   0400        LDA  DOSINI        ; SAVE DOSINI
0610 8D0106 0410        STA  INITDOS+1
0613 A50D   0420        LDA  DOSINI+1
0615 8D0206 0430        STA  INITDOS+2
0618 A900   0440        LDA  #INITDOS&255  ; SET DOSINI
061A 850C   0450        STA  DOSINI
061C A906   0460        LDA  #INITDOS/256
061E 850D   0470        STA  DOSINI+1
0620 A500   0480        LDA  NEWMEM&255    ; SET MEMLO
0622 8DE702 0490        STA  MEMLO
0625 A930   0500        LDA  #NEWMEM/256
0627 8DES02 0510        STA  MEMLO+1
062A 60     0520        RTS
062B        0530        *=   $2E2
02E2 0E06   0540        .WORD GRABDOSI     ; SET RUN ADDRESS
02E4        0550        .END
```

Figure 8-3 Reset MEMLO

INTERRUPT PROCESSING STRUCTURE

The capability to selectively respond to special hardware and software events (i.e. interrupts), provides enormous flexibility to any computer system. As in any 6502-based system, there are two types of interrupt requests at the processor level, maskable (IRQ) and nonmaskable (NMI) interrupts. A higher level of interrupt control is provided by ANTIC, POKEY and the PIA chip. Each of these chips is responsible for mediating a number of events which could cause interrupts. If a particular interrupt is enabled at the level of the three guardian chips, then they allow the interrupt request to pass on to the 6502. ANTIC handles NMI requests, and POKEY and the PIA handle IRQ requests.

The following interrupt functions are available:

Name (vector)	**Type**	**Function**	**Used by**

DISPLAY LIST	NMI	Graphics timing	User
(VDSLST)	NMI	System init.	OS
SYSTEM RESET (none)	NMI	Graphics display	OS,user
VERTICAL BLANK	IRQ	Serial input	OS
(VVBLKI,VVBLKD)	IRQ	Serial output	OS
SERIAL INPUT READY	IRQ	Serial output	OS
(VSERIN)	IRQ	Hardware timer	User
SERIAL OUTPUT	IRQ	Hardware timer	User
READY (VSEROR)	IRQ	Hardware timer	User
SERIAL OUTPUT	IRQ	Key pressed	OS
COMPLETE (VSEROC)	IRQ	[BREAK] key	OS
POKEY TIMER 1	IRQ	Device proceed	Unused
(VTIMR1)	IRQ	Device interrupt	Unused
POKEY TIMER 2			
(VTIMR2)			
*POKEY TIMER 4			
(VTIMR4)			
KEYBOARD (VKEYBD)			
*BREAK KEY (BRKKY)			
SERIAL BUS			
PROCEED (VPRCED)			
SERIAL BUS			
INTERRUPT (VINTER)			

* This IRQ is vectored only in the Rev. B version of the OS

Section 6 of the OPERATING SYSTEM Manual contains more detailed information on interrupts. Extreme care needs to be taken in working with interrupts. For example, if you accidentally disable the keyboard IRQ interrupt, the computer will ignore all the keys except the [BREAK] key. Although this may be useful sometimes, it may make debugging your program a bit difficult!

The IRQ Interrupt Handler

The OS has an IRQ interrupt handler that processes the various IRQs. This handler has RAM vectors for all of the IRQs. (Note - the [BREAK] key IRQ is not vectored in the original version of the OS The IRQ vectors are set to their initial values during both power-up and SYSTEM RESET. The locations of the IRQ RAM vectors are described in the subsection on System Vectors.

IRQ vector functions are:

VIMIRQ - Immediate IRQ vector. All IRQs vector through this location. VIMIRQ normally points to the system IRQ handier. You can steal this vector to do your own IRQ Interrupt processing.

VSEROR - Pokey Serial Output Needed IRQ vector. This normally points to the code to provide the next byte in a buffer to the serial output port.

VSERIN - Pokey Serial Input Ready IRQ vector. This points to the code to place a byte from the serial input port into a buffer.

VSEROC - Pokey Serial Output Complete IRQ vector. Normally this vector points to code that sets a transmit done flag after the checksum byte goes out.

VTIMR1 - Pokey Timer 1 IRQ vector. Initialized to point to a PLA,RTI instruction sequence.

VTIMR2 - Pokey Timer 2 IRQ vector. Initialized to point to a PLA,RTI instruction sequence.

VTIMR4 - Pokey Timer 4 IRQ vector. Initialized to point to a PLA,RTI instruction sequence.

VKEYBD - Keyboard IRQ vector. Pressing any key except [BREAK] causes this IRQ. VKEYBD can be used to preprocess the key code before it is converted to ATASCII by the OS. VKEYBD normally points to the OS keyboard IRQ routine.

BRKKY - [BREAK] key vector. In the Rev. B version of the OS, this IRQ has its own vector. It is initialized to a PLA,RTI instruction sequence.

VPRCED - Peripheral Proceed IRQ vector. The proceed line is available to peripherals on the serial bus. This IRQ is unused at present and normally points to a PLA,RTI instruction sequence.

VINTER - Peripheral interrupt IRQ vector. The interrupt line is also available on the serial bus. VINTER normally points to a PLA,RTI instruction sequence.

VBREAK - 6502 BRK instruction IRQ vector. Whenever a software break instruction is executed, this interrupt occurs. VBREAK can be used to set break points for a debugger, though it normally points to a PLA,RTI instruction sequence.

The IRQs are enabled and disabled as a group by the 6502 instructions CLI and SEI respectively. The IRQs also have individual enable/disable bits in POKEY. The programmer's reference card provided with this book shows the IRQs and their enable/disable bits.

The IRQEN register contains most of the IRQ enable/disable bits and is a write-only register. The OS keeps a shadow copy of IRQEN in POKMSK ($0010), but IRQEN is not updated from POKMSK during vertical blank. Each interrupt is enabled by setting the proper bit in IRQEN to a one. A zero is placed in a bit in IRQEN to clear interrupt status from that corresponding bit in IRQST. You might note that bit 3 in IRQST (Serial data transmission is finished) is not cleared by this process. This bit is simply a status bit and reflects the current status of the serial transmission register.

PACTL and PBCTL are used to enable and test the status of the IRQs handled by the PIA. Bit 0 of each of these registers is the interrupt enable for that port. Bit 7 represents the interrupt status. This bit is cleared whenever the PACTL or PBCTL registers are read.

Using The IRQs

The availability of the IRQ vectors means that you can tailor much of the system I/O to your liking. Currently, the OS does not provide for overlapping I/O with other processing. By redirecting the three serial I/ O interrupt vectors however, it is possible to rewrite portions of the I/O subsystem to allow for true concurrent processing.

The three timer interrupts can be put to use in any operation requiring precise timing control. These timers would normally be used when the 60-Hertz software timers are too slow. Refer to the subsection on Real Time Programming for more information on this topic.

Many applications require that programs be protected from user input error. A couple of the IRQ vectors can be used to provide extended input protection. The example in Figure 8-4 uses the VKEYBD IRQ vector to disable the control key. The routine also masks the [BREAK] key by stealing the VIMIRQ vector and ignoring the [BREAK] key interrupt. Though written for the original version of the OS, this routine will still work in Rev. B.

Two of the IRQs are handled by the PIA, VPRCED and VINTER. These are unused by the OS, and may be utilized to provide more control over external devices.

```
0010         10 POKMSK   =    $0010
D209         20 KBCODE   =    $D209
0208         30 VKEYBD   =    $0208
D20E         40 IRQEN    =    $D20E
D20E         45 IRQST    =    IRQEN
0216         46 VMIRQ    =    $0216
0000         60          *=   $600
0600 78      80 START    SEI                      DISABLE IRQS
0601 AD1602 90           LDA     VMIRQ            REPLACE THE IRQ VECTOR
0604 8D4D06 0100         STA     NBRK+1           WITH OUR OWN
0607 AD1702 0110         LDA     VMIRQ+1          ALL IRQS WILL
060A 8D4E06 0120         STA     NBRK+2           GO TO NBRK
060D A945   0130         LDA     #IRQ&255
060F 8Dl602 0140         STA     VMIRQ
0612 A906   0150         LDA     #IRQ/256
```

```
0614 8Dl702 0160        STA     VMIRQ+1
0617 AD0802 0200        LDA     VKEYBD      POINT KEY IRQ TO
061B 8D4306 0210        STA     JUMP+1      REP
061E AD0902 0220        LDA     VKEYBD+1
0621 8D4406 0230        STA     JUMP+2
0624 A939   0240        LDA     #REP&255    VECTOR KEY IRQ
0626 8D0802 0250        STA     VKEYBD      LOW BYTE OF VECTOR
0629 A906   0260        LDA     #REP/256
062B 8D0902 0270        STA     VKEYBD+1
062E 58     0170        CLI                 ENABLE IRQS
062F 60     0280        RTS
            0290        *=$639
0639 AD09D2 0300 REP    LDA                 KBCODE ALL KEY IRQS COME HERE
063C 2980   0310        AND     #$80        CHECK IF CONTROL HIT
063E F002   0320        BEQ     JUMP        IF NOT HIT THEN GO
0640 68     0330        PLA                 ELSE IGNORE CONTROL KEY
0641 40     0340        RTI
0642 4C4206 0360 JUMP   JMP                 JUMP THIS CALLS THE OLD KEY IRQ
0645 48     0375 IRQ    PHA                 ALL IRQS COME HERE
0646 AD0ED2 0380        LDA     IRQST       CHECK IF [BREAK]
0649 1004   0390        BPL     BREAK       IF [BREAK] IRQ,BRANCH
064B 68     0405        PLA                 ELSE CALL OLD IRQ VECTOR
064C 4C4C06 0410 NBRK   JMP     NBRK        CALL OLD IRQ VECTOR
064F A97F   0430 BREAK  LDA     #$7F        HERE IF [BREAK]
0651 8D0ED2 0440        STA     IRQST       SHOW NO [BREAK]
0654 A510   0450        LDA     POKMSK
0656 8D0ED2 0460        STA     IRQEN
0659 68     0462        PLA
065A 40     0464        RTI                 RETURN AS IF NO [BREAK]
065B        0470        *=   $02E2
02E2 0006   0480        .WORD START
```

Figure 8-4 Protecting Programs from User Input Error

The NMI Handler

The OS has an NMI handler for handling the nonmaskable interrupts. Unlike the IRQs, the NMIs cannot be "masked" (disabled) at the 6502 level. All the NMIs except SYSTEM RESET can be disabled by ANTIC.

Two of the NMIs, the display list interrupt (DLI) and the vertical blank (VBLANK) interrupt, have RAM vectors that can be used. In fact, VBLANK can be intercepted in two places, immediate or deferred VBLANK. The NMI vectors are:

Name	**Vector**
SYSTEM RESET	none
DISPLAY LIST INTERRUPT	VDSLST ($0200)
VERTICAL BLANK	
IMMEDIATE	VVBLKI ($0222)
DEFERRED	VVBLKD ($0224)

The SYSTEM RESET NMI doesn't have a RAM vector. SYSTEM RESET always results in a jump to the monitor warmstart routine. However.- the DOSINI RAM vector is used during the Warmstart process, and thus can be used to trap the [SYSTEM RESET] (see subsection on the Monitor).

The DLI vector is unused by the OS. Refer to Section 5, Display List Interrupts, for details on using this feature.

Vertical Blank Interrupt Processing

The vertical blank interrupt facility is an extremely valuable resource to a programmer. These interrupts are nonmaskable and occur at regular intervals based on the television signal standard (every 60th of a second for NTSC, every 50th for PAL). Just as importantly, these interrupts occur during that period of time when the screen has been blanked, so that changes made during this period will not be immediately seen on the display. This leads to a wide assortment of uses.

After a vertical blank interrupt has been recognized by the OS, the contents of the A, X, and Y registers are placed onto the stack. The system then vectors through the immediate vertical blank

vector (VVBLKI) located at $0222. This vector normally points to the OS vertical blank interrupt service routine at $E45F. The OS uses this routine to increment the real time clock, to decrement the system timers, to do color attracting, to copy shadow registers,. and to update values from the input controllers. This routine terminates by jumping through the deferred vertical blank vector (VVBLKD) at $0224. This vector is initialized to point to a simple interrupt termination routine at $E462. Figure 8-5 illustrates this process.

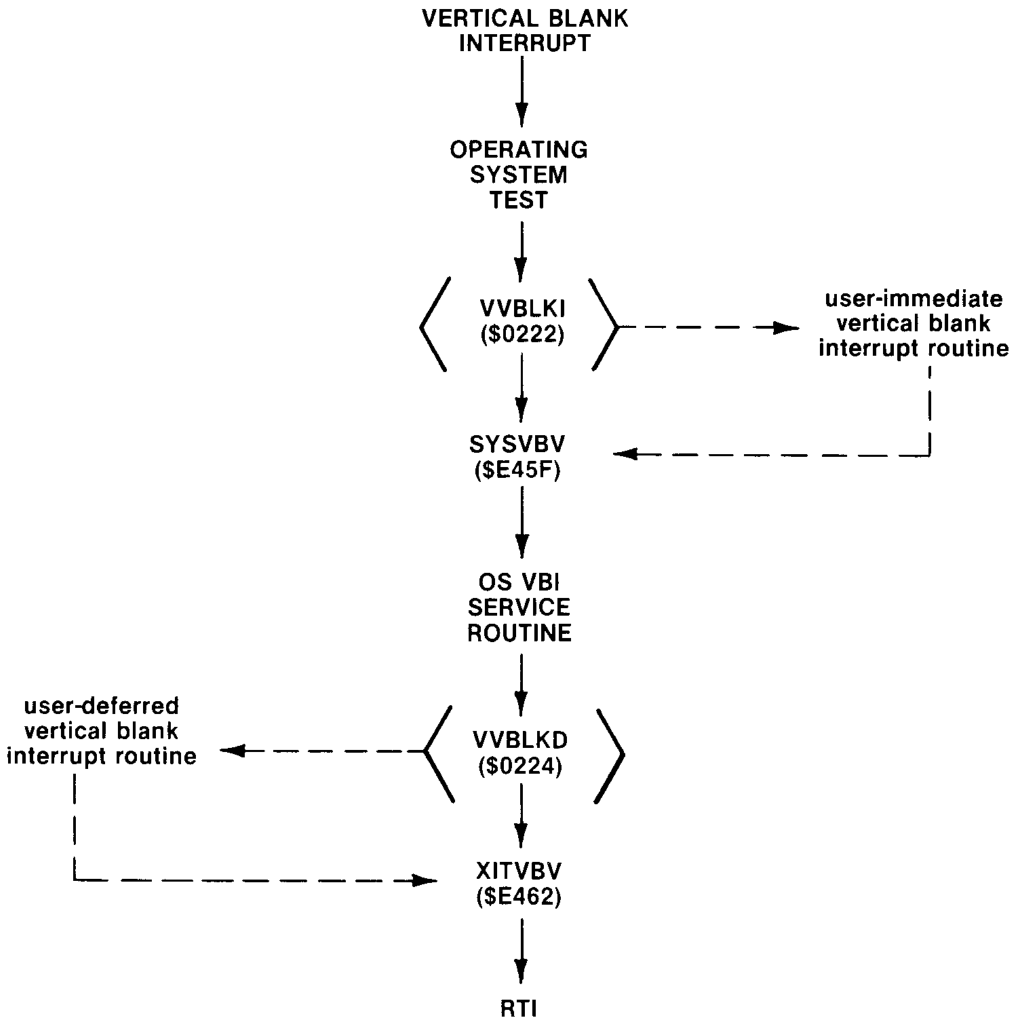


Figure 8-5 Vertical Blank Interrupt Execution

These two vectors were put into RAM to allow programmers to trap the vertical blank service routine and use the 60-Hertz interrupt for their own purposes. The procedure to use them is straightforward. First decide whether the vertical blank interrupt (VBI) routine is to be an immediate VBI or a deferred VBI. In many cases it makes little difference which is chosen. There are some instances where it does matter. The first case arises when your VBI routine reads or writes to registers which are shadowed by the OS VBI routine. It might be necessary to write to the hardware registers after the OS VBI has written to them. He who writes last, writes best!

The second case arises when your VBI routine consumes too much processor time. The OS VBI routine may then be delayed beyond the end of the vertical blank period. This in turn may cause some graphics registers to be changed while the beam is tracing on the screen. The result may be unsightly. if this is the case, your VBI routine should be placed as a deferred VBI routine. The time limit for immediate VBI is about 2000 machine cycles; for deferred VBI it is about 20,000 cycles.

However, many of these 20,000 machine cycles are executed while the electron beam is being drawn, so graphics operations should not be executed in extremely long deferred VBI routines. Furthermore, display list interrupt execution time comes out of the time available for this processing. Remember, VBI processing will cut down on the time available for mainline code execution.

The third case arises when your own VBI must be mixed with time- critical I/O such as to the disk or cassette. The OS immediate VBI routine has two stages, a critical and a non-critical stage. During time critical I/O, the OS immediate VBI routine aborts after stage one processing is complete. if you want your VBI to be executed during every vertical blank period, it must be defined as an immediate VBI. Be wary in this situation, for this may result in interference problems with time-critical I/O.

Once you have decided whether your VBI routine should be immediate or deferred, you must place the routine in memory (page six is a good place), link it to the proper VBI routine, and modify the appropriate OS RAM vector to point to it. Terminate an immediate VBI routine with a JMP to $E45F. Terminate a deferred VBI routine with a JMP to $E462. If you want to bypass the OS VBI routine entirely (and thereby gain processing time), terminate your immediate VBI routine with a JMP to $E462.

A common problem with interrupts on 8-bit micros arises when you try to change the vector to the interrupt. Vectors are 2-byte quantities; it takes two store instructions to change them. There is a small chance that an interrupt will occur after the first byte has been changed but before the second byte has been updated. This could crash the system. The solution to this problem has been provided by an OS routine called SETVBV at $E45C. Load the 6502 Y- register with the low byte of the address, the X-register with the high byte of the address, and the accumulator with a 6 for immediate VBI or a 7 for deferred VBI. Then JSR SETVBV and the interrupt will be safely enabled. The new VBI routine will begin executing within one 60th of a second.

A wide variety of operations can be performed with 60-Hertz interrupts. First, screen manipulations can be done during the vertical blank to ensure that transitions do not occur on the screen. Second, high speed regular screen manipulations can be performed. This can be very important in rhythmic animations where changes need to occur at a pace independent from other processing.

Another function of vertical blank interrupts is for real-time sound effects. The sound registers in the ATARI 400/800 allow control of

frequency, volume, and distortion, but not duration. Duration can be controlled with a VBI by having a calling routine set some duration parameter which the VBI then decrements down to zero. This technique can be used to control the volume of the sound and so give attack and decay envelopes to sounds. Finer control of frequency and distortion is possible with extensions of this technique. The result can be very intricate sound effects. Since the VBI time resolution is only 1/60th of a second, VBls are not useful for the volume only mode of the audio channels.

VBls are also useful for handling user inputs. Usually, these inputs require little processing, but constant attention. A VBI allows the program to check for user input every 60th of a second without burdening the' program. It is an ideal solution to the problem of maintaining computational continuity without ignoring the user.

Finally, VBls allow a crude form of multitasking to take place. A foreground program can run under the VBI while a background program runs in the mainline code. As with any interrupt, careful separation of the databases for the two programs must be maintained. The power derived from the vertical blank interrupt is, however, well worth the effort.

THE SYSTEM VECTORS

One measure of the power of any operating system is its adaptability. Just how easy is it for a user to take advantage of the OS routines or modify and customize system routines?

In this regard, the OS for the ATARI Home Computer System would score well. In practically every instance where access to system routines could be beneficial, the OS has "hooks" where you can attach or replace system routines with your own.

This flexibility is provided by a combination of several different mechanisms. The first of these is a ROM table of JMP instructions to crucial OS routines. In future versions of the OS, the location of this Jump table will not change, although the values there probably will. Thus, if your software accesses the main OS routines via this table, it will also work on future versions of the OS. If your software does not use these ROM vectors, but instead jumps directly into the OS ROM, then it will almost certainly crash on future versions of the OS.

The second mechanism is a series of address vectors in RAM which link many of the interrupt processing routines together. To alter the handling of a particular interrupt, one need change only a single vector to point to the replacement code. The OS initializes these vectors as part of the power-up sequence. Again, though the initialized contents of these vectors may change, their location is guaranteed not to.

The third mechanism is the device handler table where vectors to specific device handlers (e.g. disk drive, printer,...) are stored. For a discussion of this facility, refer to the Centralized Input/Output subsection of this section.

Name	**Location**	**Use**
DISKIV	$E450	Disk handler initialization
DSKINV	$E453	Disk handler vector
CIOV	$E456	Central I/O routine vector
SIOV	$E459	Serial I/O routine vector
SETVBV	$E45C	Set system timers routine vector
SYSVBV	$E45F	System vertical blank processing
XITVBV	$E462	Exit vertical blank processing
SIOINV	$E465	Serial I/O initialization
SENDEV	$E468	Serial bus send enable routine
INTINV	$E46B	Interrupt handler routine
CIOINV	$E46E	Central I/O initialization
BLKBDV	$E471	Blackboard mode (Memopad) vector
WARMSV	$E474	Warm start entry point (SYSTEM RESET)
COLDSV	$E477	Cold start entry point (power-up)
RBLOKV	$E47A	Cassette read block routine vector
CSOPIV	$E47D	Cassette open for input vector

Figure 8-6 ROM Jump Vectors

Since this ROM table is actually a table of three byte JUMP instructions, an example of using a ROM vector is:

```
JSR CIOV
```

RAM VECTORS

Name	Location	Value	Use
-- Page Two Locations --			
VDSLST	$0200	$E7B3	Display List Interrupt NMI Vector
VPRCED	$0202	$E7B3	Proceed Line IRQ Vector -- Unused at present
VINTER	$0204	$E7B3	Interrupt Line IRQ Vector -- Unused at Present
VBREAK	$0206	$E7B3	Software Break Instruction IRQ Vector
VKEYBD	$0208	$FFBE	Keyboard IRQ Vector
VSERIN	$020A	$EB11	Serial Input Ready IRQ Vector
VSEROR	$020C	$EA90	Serial Output Ready IRQ Vector
VSEROC	$020E	$EAD1	Serial Output Complete IRQ Vector
VTIMR1	$0210	$E7B3	POKEY Timer 1 IRQ Vector
VTIMR2	$0212	$E7B3	POKEY Timer 2 IRQ Vector
VTIMR4	$0214	$E7B3	POKEY Timer 4 IRQ Vector
VIMIRQ	$0216	$E6F6	Vector to IRQ Handler
VVBLKI	$0222	$E7D1	Immediate Vertical Blank NMI Vector
VVBLKD	$0224	$E93E	Deferred Vertical Blank Vector
CDTMA1	$0226	$xxxx	System Timer 1 JSR Address
CDTMA2	$0228	$xxxx	System Timer 2 JSR Address
BRKKY	$0236	$E754	BREAK key Vector (** only Rev. B **)
-- Page Zero Locations --			
CASINI	$0002	$xxxx	Vector for bootable cassette program initialization
DOSINI	$000C	$xxxx	Disk Initialization Vector
DOSVEC	$000A	$xxxx	Disk Software Run Vector
RUNVEC	$02E0	$xxxx	DUP File Load and GO Run Vector
INIVEC	$02E2	$xxxx	DUP File Load and Go Initialization Vector

Figure 8-7 RAM Vectors

Unlike the ROM Jump tables, these vectors are true two byte address vectors. A typical calling sequence to use one of the RAM vectors might look like this:

```
     JSR CALL
CALL JMP (DOSINI)
```

THE CENTRALIZED INPUT/OUTPUT SUBSYSTEM

One of the most taxing problems facing an operating system designer is how to handle input/output to the wide variety of peripherals that might be hooked up to the system. A few general philosophical guidelines to efficient I/O handling are:

- The transfer of data should be device-Independent.

- The I/O structure should support single-byte, multiple-byte, and record-organized data transfers.

- Multiple devices/files should be concurrently accessible.

- Error handling should be device-transparent.

- The addition of new device handlers should be possible without having to change the OS ROM.

The ATARI 400/800 OPERATING SYSTEM (OS) was designed to provide exactly these capabilities. The ATARI 400/800 OS uses a table- driven centralized input/output subsystem. From the OS standpoint, I/O is organized around the IOCB, or Input/Output Control Block. An IOCB is a standard table that specifies one complete input or output operation. Any of eight standard I/O operations can be executed through an IOCB. By changing an entry in the IOCB, the user can change the nature of the input/output operation, even the physical device which is the target of the I/O. Thus, a user can easily perform I/O to completely different devices such as the disk drive and the printer without having to worry about hardware details. Most I/O requires only that the user set up an IOCB with control data, and then pass control to the I/O subsystem.

Two types of elements make up the I/O subsystem: I/O system routines and I/O system control blocks. The I/O system routines include the central I/O routine (CIO), the device handlers (E:, K:, S:, P:, C:, D:, R:) and the serial I/O routine (SIO). The Handler Address Table (HATABS) plays a major role in linking CIO with the individual device handlers. The system I/O control blocks contain control data that is routed to the I/O subsystem. The user interface is the same for all devices (e.g., the commands to output a line to the printer (P:) or to the display editor (E:) are very similar).

Understanding the structure of the I/O subsystem will enhance your use of it. Figure 8-8 shows the relationship of the I/O system routines and the I/O system control blocks.

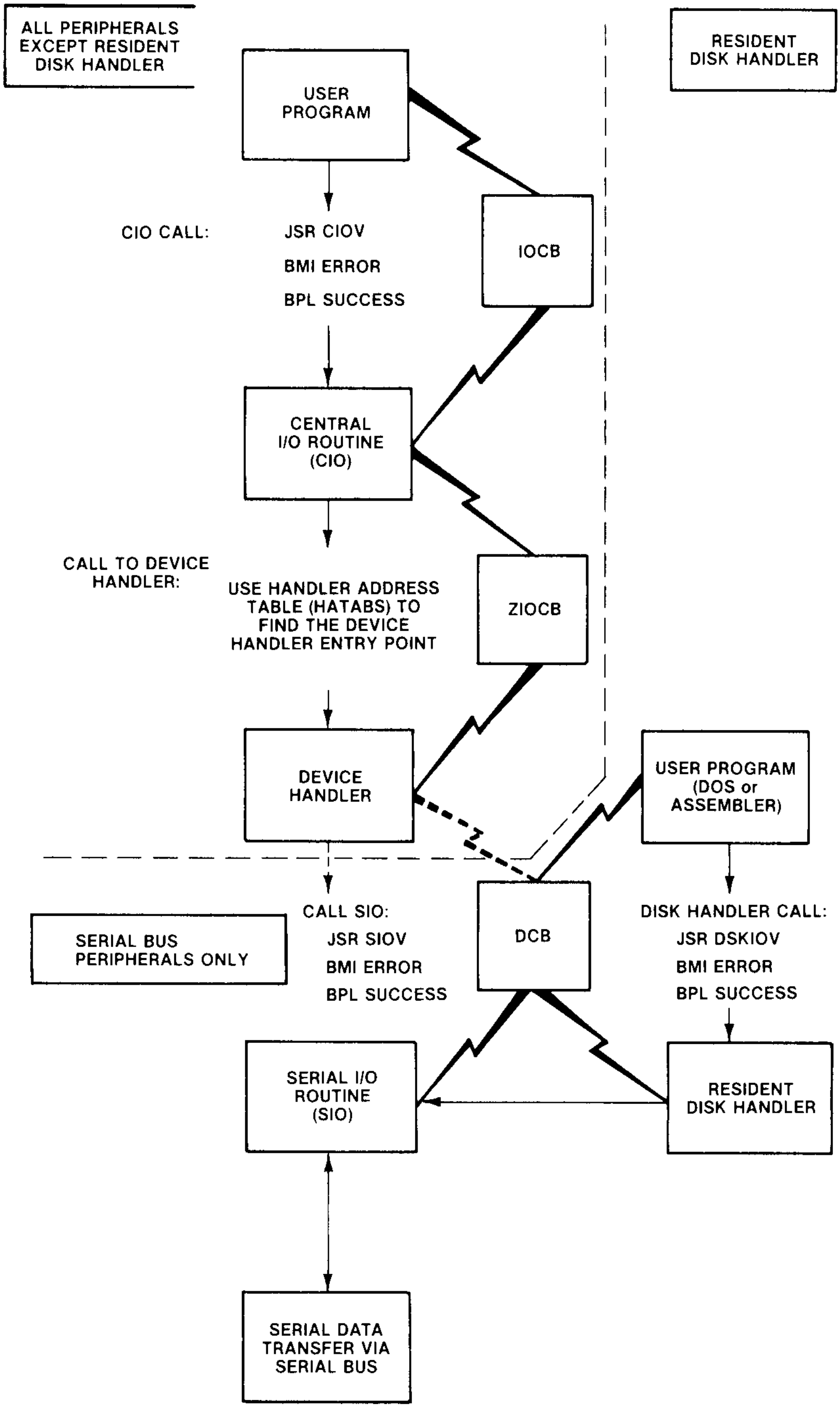

ALL PERIPHERALS EXCEPT RESIDENT DISK HANDLER
RESIDENT DISK HANDLER
USER PROGRAM
CIO CALL:
JSR CIOV
BMI ERROR
BPL SUCCESS
IOCB
CENTRAL I/O ROUTINE (CIO)
CALL TO DEVICE HANDLER:
USE HANDLER ADDRESS TABLE (HATABS) TO FIND THE DEVICE HANDLER ENTRY POINT
ZIOCB
DEVICE HANDLER
USER PROGRAM (DOS or ASSEMBLER)
SERIAL BUS PERIPHERALS ONLY
CALL SIO:
JSR SIOV
BMI ERROR
BPL SUCCESS
DCB
DISK HANDLER CALL:
JSR DSKIOV
BMI ERROR
BPL SUCCESS
SERIAL I/O ROUTINE (SIO)
RESIDENT DISK HANDLER
SERIAL DATA TRANSFER VIA SERIAL BUS

Figure 8-8 I/O Subsystem

I/O System Control Blocks

There are four types of control blocks:

-Input/Output Control Block (IOCB)
-Zero-Page I/O Control Block (ZIOCB)
-Device Control Block (DCB)
-Command Frame Buffer (CFB)

The I/O system control blocks are used to communicate information about the I/O function to be executed. The control blocks provide the I/ O system routines with control information to perform the I/O function. Refer to the OPERATING SYSTEM Manual for information as to the detailed structure of the four types of control blocks.

Eight IOCBs in the OS are used to effect communication between user programs and CIO. Figure 8-9 shows the content of an IOCB for some common

I/O functions. The IOCBs are:

Name	Location, Length
IOCB0	[$340,16]
IOCB1	[$350,16]
IOCB2	[$360,16]
IOCB3	[$370,16]
IOCB4	[$380,16]
IOCB5	[$390,16]
IOCB6	[$3A0,16]
IOCB7	[$3B0,16]

A second type of control block, the ZIOCB [$0020,16], is used to communicate I/O control data between CIO and the device handlers. When called, CIO uses the value contained in the X-register as an index to which of the IOCBs (one of eight) is to be used. CIO then moves the control data from the selected IOCB to the ZIOCB for use by the appropriate device handler. The ZIOCB is of little interest unless you are writing a new device handler or are replacing a current one.

Device handlers that require I/O over the serial bus will then load control information into the DCB [$0300,12]. SIO will use the DCB information and return the status information in the DCB for subsequent use by the device handler. Figure 8-10 illustrates some common I/O functions and the contents of their associated DCBs.

The resident disk handler does not conform to the regular user-CIO- handler-SIO calling sequence. Instead, you use the DCB to communicate directly with the resident disk handler. Section 9, the Disk Operating System, contains more information on the resident disk handler.

The last type of control block encountered in the I/O subsystem is the Command Frame Buffer (CFB). This 4-byte table located at $023A is used by the SIO routine while performing the serial bus operations. The four bytes contain the device code, command code, and command auxiliary bytes 1 and 2. The data buffer that is transmitted is defined by two pointers BUFRLO [$0032,2] and BFENLO [$0034,2]. In general it is not recommended that the OS be used at this level. Other parameters have to be set, and extreme care must be taken in calling the proper sequence of subroutines. CIO and SIO were designed to be easily called by user programs. Use them --- but stay away from the command frame buffer!

IOCB CHART

Call	ICHID	ICDNO	ICCOM	ICSTA	ICBAL	ICBAH	ICPTL	ICPTH	ICBLL	ICBLH	ICAX1	ICAX2
OPEN	X	X	3	note 1	$80	06	X	X	X	X	4	0
FILE-R	X	X	3	note 1	$80	06	X	X	X	X	8	note 2
EAD	X	X	7	note 1	00	06	X	X	$80	00	X	X
OPEN	X	X	$B	note 1	00	06	X	X	$80	00	X	X
FILE-	X	X	5	note 1	00	06	X	X	$80	00	X	X
WRIT	X	X	9	note 1	00	06	X	X	$80	00	X	X
E	X	X	$C	note 1	X	X	X	X	X	X	X	X
GET	X	X	$D	note 1	X	X	X	X	X	X	X	X
BYTES												
PUT												
BYTES												
GET												
RECO												
RD												
PUT												
RECO												
RD												
CLOS												
E FILE												
STAT												
US												

Note 1=The status of the I/O command is stored here and in the Y REG. on return from CIO.

Note 2=The Auxilary bytes of the IOCB's are used by some handlers to indicate special modes.

X = Indicates ignore but do not change the current value.

GENERAL NOTE: The above IOCB definitions assume:

```
*=$600
IOBUFF .RES 80       USER I/O BUFFER
FILE  .BYTE 'D:MYPROG.BAS' USER  FILENAME
```

Figure 8-9 Input Output Control Block (IOCB)

			Disk 810/815						
			Read Sector		Write Sector				
Function	**Name**	**Location**	**810**	**815**	**810**	**815**	**Put**	**Format**	**Printer 820 Write**
Serial	DDEVIC	[$0300]	$30	$30	$30	$30	$30	$30	$30
Bus I.D.	DUNIT	[$0301]	1-4	1-8	1-4	1-8	1-4	1-4	1
Device	DCOMN	[$0302]	$52	$52	$57	$57	$50	$21	$57
Number	D	[$0303]	$40	$40	$80	$80	$80	$40	$80
Comman	DSTATS	[$0304]	U	U	U	U	U	U	U
d Byte	DBUFLO	[$0305]	U	U	U	U	U	U	U
Status	DBUFHI	[$0306]	$30	$30	$30	$30	$31	$130	5
Buffer	DTIMLO	[$0308]	$80	00	$80	00	$80/00	-	$40
Address	DBYTLO	[$0309]	$00	01	$00	01	$00/01	-	$00
	DBYTHI	[$030A]	2*	2*	2*	2*	- 2*	-	1*
Device	DAUX1	[$030B]	2*	2*	2*	2*	- 2*	-	1*
Timeout	DAUX2								
Buffer									
Length									

1* = This bytes determines printer mode (see 820 manual).

2* = DAUX1 + DAUX2 specify sector for READ, WRITE (PUT), or WRITE verify.

U = Indicates user-set address

- = Indicates ignored

Figure 8-10 Device Control Block (DCB)

Central I/O System Routine

The main function of CIO is to take control data from an IOCB and ensure that it is routed to the specific device handler needed, and then to pass control to that handler. CIO also acts as a pipeline for most I/O in the system. Most of the OS I/O functions use CIO as a common entry point, and all handlers exit via CIO. For example, BASIC will call CIO when performing an OPEN or a GRAPHICS statement. CIO supports the following functions:

OPEN	Device/file open
CLOSE	Device/file close
GET CHARS	Read N characters
READ RECORD	Read next record
PUT CHARS	Write N characters
WRITE RECORD	Write next record
STATUS	Get device status
SPECIAL	Device handler specific (e.g., NOTE for FMS)

You may wish to make your own CIO calls. The calling sequence for CIO is:

```
               ;rem user has already set up IOCB
LDX #IOCBNUM   ;set the IOCB index (IOCB * 16)
JSR CIOV       ;system routine vector to CIO
BMI ERROR      if branch taken then CIO returned
               ;error code in Y-register
```

As shown in the above call, one of the IOCBs is used to communicate control data to CIO. You may use any one of the eight IOCBS. CIO expects the IOCB index to be in the X-register. Note that this value must be the IOCB number multiplied by 16. The reason is that CIO uses this value as a true index into the various IOCBs, and each IOCB is 16 bytes. Upon return, the sign bit of the 6502 is set to indicate success or error in the I/O operation. If the N-bit is clear (i.e., a zero) the I/O was done successfully, and the Y-register will contain a 1. If the N-bit is set, the I/O request resulted in an error; in that case the Y-register will contain the error code number. A BMI instruction to an error routine is the usual way to test for operational success. The error/success value is also returned in the IOCB byte ICSTA (see IOCB definition). Chapter 5 of the OPERATING SYSTEM Manual has a sample program segment that calls CIO to OPEN a disk file, READ some records., then CLOSE the file.

CIO copies the control data from the selected IOCB to the page zero ZIOCB. CIO then determines the device handler entry point and vectors to the appropriate device handler routine. Figure 8-11 is a flowchart of the CIO system routine.

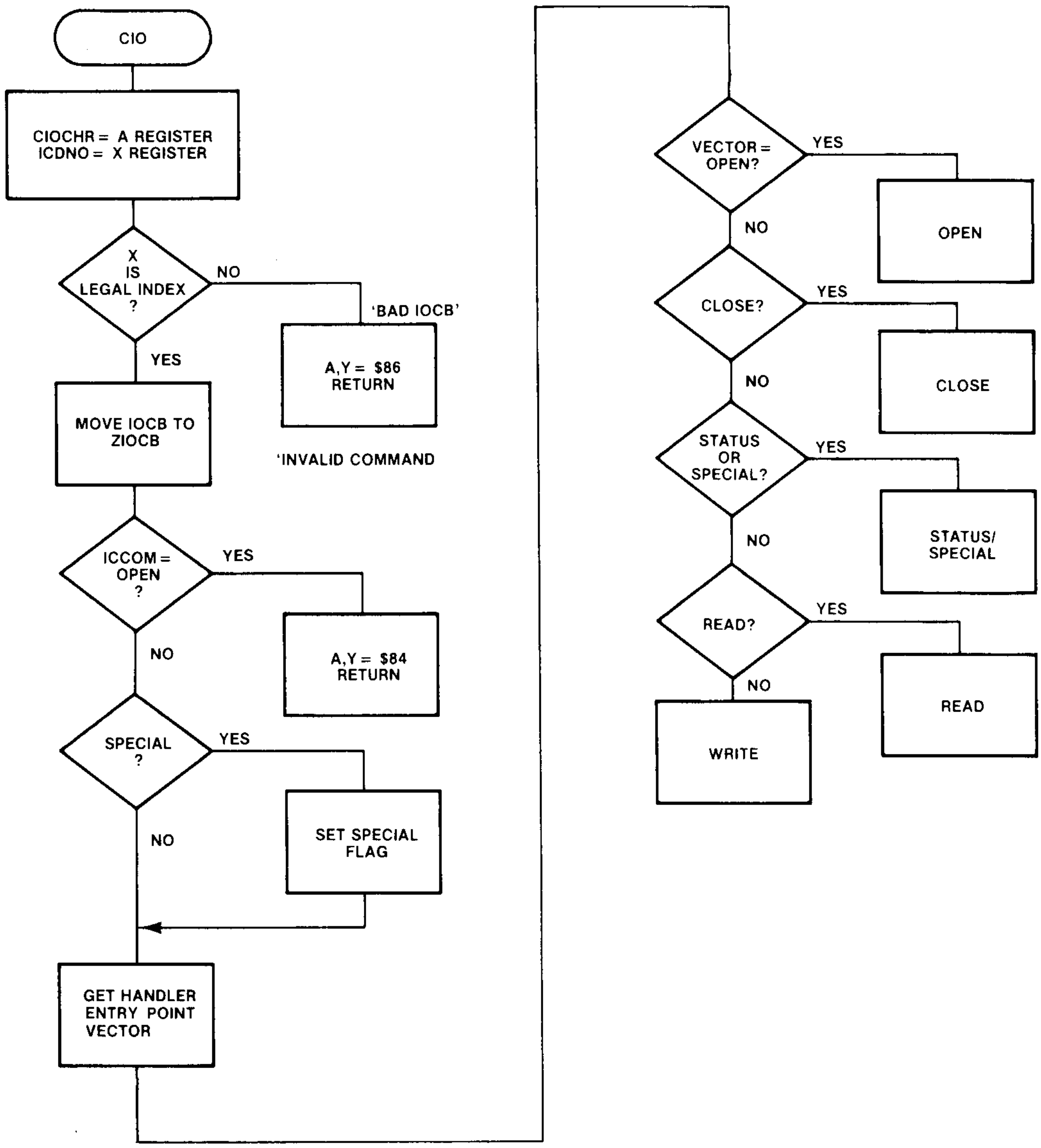


Figure 8-11.1 CIO Routine

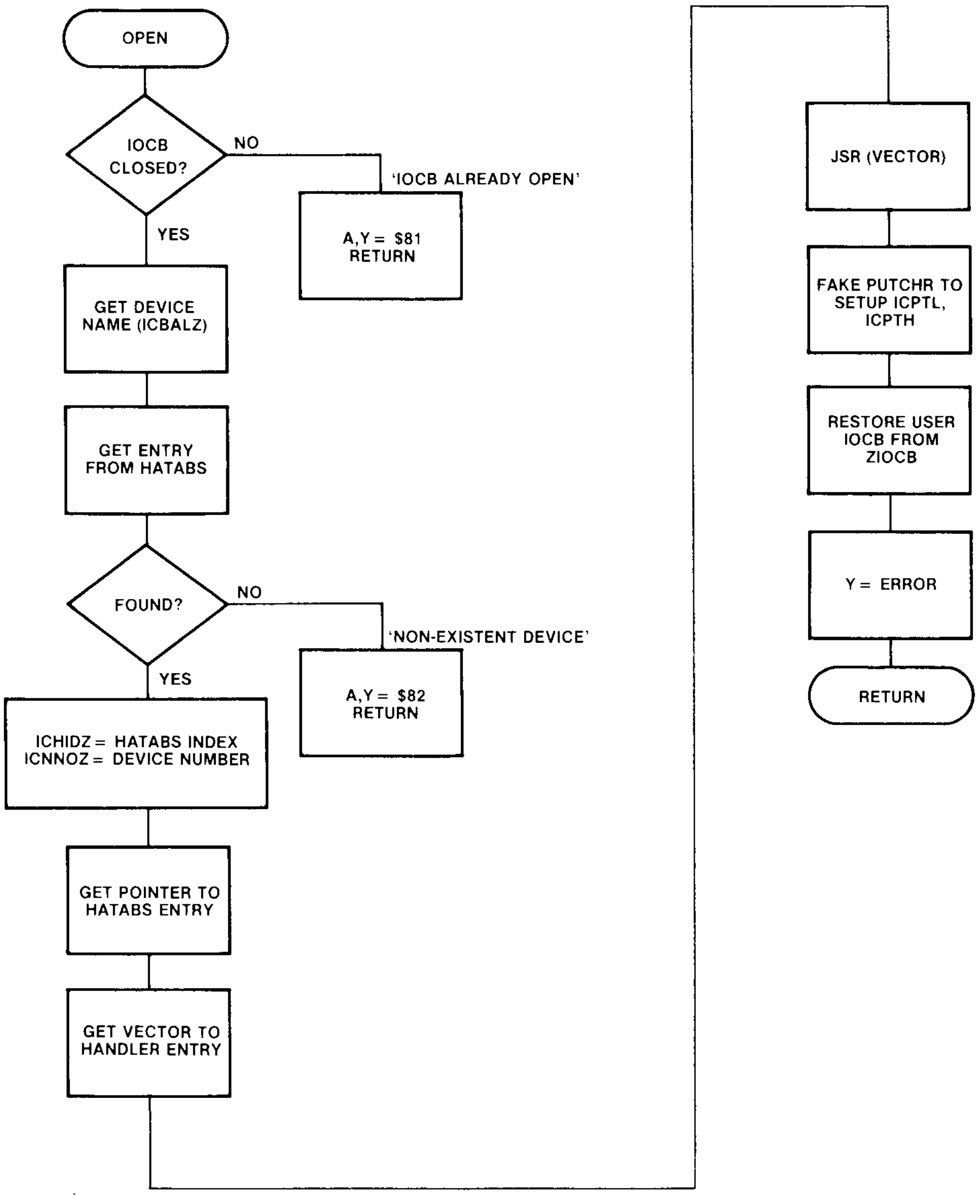


Figure 8-11.2 CIO Routine

The Handler Address Table

CIO calculates the device handier entry point in an indirect manner. First of all, an OPEN call must preceed any other I/O function to a device. While processing an OPEN command, CIO retrieves the device specification for the file being opened. The device specification is an ATASCII string pointed to by the buffer address portion of the IOCB. The first element in this string must be a one character device identifier (e.g., `D' for disk drive, `Pt for printer,...). CIO then searches for this character in a table of handier entry points called HATABS, which runs from $031A to $033B (Figure 8-12 shows the layout of HATABS). CIO begins the search at the bottom of HATABS and works upward until a match is found for the device identifier. The search is performed in this direction to facilitate the addition or modification of device handlers. During the initialization code, the HATABS table is copied from ROM down into RAM. Devices that are then booted (e.g. the disk drive, or RS-232 module) then add their handler information to the bottom of the table. There is room in the table for a total of 14 entries, 5 of which are set up during system initialization. If a new

printer handier were added to the bottom of the table, CIO would find it before the one transferred from ROM. This allows new handlers to replace old ones.

After the device identifier has been located, CIO knows that the next two bytes point to the devices' entry point table. This is a table of addresses for the routines that handle each of the CIO functions. Figure 8-13 gives the layout of a typical entry point table.

To find which one of the handier entry points to vector through, CIO then uses ICCOM, the IOCB command byte, as an index into the entry table. The entry point tables for all of the resident device handlers can be found in the OS listing. The relative position of each of the vectors in an entry table have the same meaning from table to table. For example, the first position in all of the device handler entry point tables is the vector to the device handler OPEN routine.

You can take advantage of the flexibility of HATABS to add some new features to the OS. Figure 8-14 is an example of how to add a null handler. A null handler does exactly what its name implies: nothing. This can be useful in debugging programs. Instead of waiting around for 50,000 disk accesses to find a bug, output can be directed to the null handier. With a null handler, trouble spots in programs can be identified more quickly.

```
             01 ; HANDLER ADDRESS TABLE
E430         02 PRINTV  =     $E430
E440         03 CASETV  =     $E440
E400         04 EDITRV  =     $E400
E410         05 SCRENV  =     $E410
E420         06 KEYBDV  =     $E420
             07 ;
0000         08         *=   $031A
             09 ;
             10 HATABS
031A 50      20         .BYTE   'P'     PRINTER
031B 30E4    30         .WORD   PRINTV                  ENTRY POINT TABLE
031D 43      40         .BYTE   'C'     CASSETTE
031E 40E4    50         .WORD   CASETV     ENTRY POINT TABLE
0320 45      60         .BYTE   'E'     DISPLAY EDITOR
0321 00E4    70         .WORD   EDITRV     ENTRY POINT TABLE
0323 53      80         .BYTE   'S'     SCREEN HANDLER
0324 10E4    90         .WORD   SCRENV     ENTRY POINT TABLE
0326 4B      0100       .BYTE   'K'     KEYBOARD
0327 20E4    0110       .WORD   KEYBDV     ENTRY POINT TABLE
0329 00      0120       .BYTE   0       FREE ENTRY 1 (DOS)
032A 00 00   0130       .BYTE   0,0
032C 00      0140       .BYTE   0       FREE ENTRY 2 (850 MODULE)
032D 00 00   0150       .BYTE   0,0
032F 00      0160       .BYTE   0       FREE ENTRY 3
0330 00 00   0170       .BYTE   0,0
0332 60      0180       .BYTE   0       FREE ENTRY 4
0333 00 00   0190       .BYTE   0,0
0335 00      0200       .BYTE   0       FREE ENTRY 5
0336 00 00   0210       .BYTE   0,0
0338 00      0220       .BYTE   0       FREE ENTRY 6
0339 00 00   0230       .BYTE   0,0
033B 00      0240       .BYTE   0       FREE ENTRY 7
```

Figure 8-12 Handier Address Table (HATABS)

```
        *=$PRINTV
E430 9E EE      .WORD  PHOPEN-1     DEVICE OPEN
E432 DB EE      .WORD  PHCLOS-1     DEVICE CLOSE
E434 9D EE      .WORD  BADST-1      DEVICE READ-NOT IMPLEMENTED
E436 A6 EE      .WORD  PHWRIT-1     DEVICE WRITE
E438 80 EE      .WORD  PHSTAT-1     DEVICE STATUS
E43A 9D EE      .WORD  BADST-1      SPECIAL-NOT IMPLEMENTED
E43C 4C 78 EE     JMP  PHINIT       DEVICE INITIALIZATION
```

Figure 8-13 Printer Handler Entry Point Table

```
0000        10           *  =    $600
031A        20 HATABS       =    $031A
0600 A000   40 START       LDY   #0
0602 B91A03 60 LOOP        LDA   HATABS,Y
0605 C900   70             CMP   #0        ;   FREE ENTRY?
0607 F009   80             BEQ   FOUND
```

```
0609 C8      90           INY
060A C8      0100         INY
060B C8      0110         INY                  ;    POINT TO NEXT HATABS ENTRY
060C C022    0120         CPY    #34           ;    AT END OF HATABS?
060E D0F2    0130         BNE    LOOP          ;    NO ... CONTINUE
0610 38      0140         SEC                  ;    YES... INDICATE ERROR
0611 60      0150         RTS
             0160 ;
0612 A94E    0180 FOUND   LDA    #'N           ;    SET DEVICE NAME
0614 991A03  0190         STA    HATABS,Y
0617 C8      0200         INY
0618 A924    0210         LDA    #NULLTAB&255
061A 991A03  0220         STA    HATABS,Y      ;    HANDLER ADDRESS
061D C8      0230         INY
061E A906    0240         LDA    #NULLTAB/256
0620 991A03  0250         STA    HATABS,Y
0623 60      0260         RTS
             0270
0624 3206    0290 NULLTAB .WORD  RTHAND-1      ;    OPEN
0626 3206    0300         .WORD  RTHAND-1      ;    CLOSE
0628 3406    0310         .WORD  NOFUNC-1      ;    READ
062A 3206    0320         .WORD  RTHAND-1      ;    WRITE
062C 3206    0330         .WORD  RTHAND-1      ;    STATUS
062E 3406    0340         .WORD  NOFUNC-1      ;    SPECIAL
0630 4C3306  0350         JMP    RTHAND        ;    INITILIZATION
             0360 ;
0633 A001    0380 RTHAND  LDY    #1            ;    SUCCESSFUL I/O FUNCTION
0635 60      0400 NOFUNC  RTS                  ;    FUNCTION NOT IMPLEMENTED
```

Figure 8-14 Null Handler

The Device Handlers

The device handlers can be divided into resident and nonresident handlers. The resident handlers are present in the OS ROM, and may be called through CIO whenever the handier has an entry in HATABS. The nonresident handlers must first be loaded into RAM and have their entry placed into HATABS before they can be called from CIO. The resident device handlers are:

(E:) Display Editor
(S:) Screen
(K:) Keyboard
(P:) Printer
(C:) Cassette

Although the nonresident handlers are not present in the OS ROM, nonresident handlers may be added by the OS during power-up or SYSTEM RESET. You can even add your own device handier during program execution. Figure 8-14 presents an example of adding a handier to the OS.

The device handlers use I/O control data passed by CIO in the ZIOCB. Data in the ZIOCB is used to perform I/O functions such as OPEN, CLOSE, PUT, and GET. Not all of the device handlers support all the I/O commands (e.g., trying to PUT a character to the keyboard results in an Error 146, Function Not implemented). Section 5 of the OPERATING SYSTEM Manual contains a list of the functions supported by each device handler, as well as complete operational details of the handlers.

Serial I/O Routine

SIO handles serial bus communication between the serial device handlers in the computer and the serial bus devices. It communicates with its caller through the device control block (DCB). SIO uses the I/ O control data in the DCB to send and receive commands and data over the serial bus. The calling sequence is:

```
          ;caller has set up the DCB to do function
JSR SIOV  ;system vector to SIO
BMI ERROR ;N-bit set indicates error in I/O execution
```

The DCB contains I/O control information for SIO and must be setup before the call to SIO. Figure 8-10 shows the contents of the DCB for some common I/O operations.

To send commands to SIO, you need to understand the structure of the DCB, which is described in Section 9 of the OPERATING SYSTEM Manual. Figure 8-15 demonstrates a simple assembly language routine to output a line to the printer by setting up the DCB and calling SIO.

```
0000        05            *= $3000 ARBITRARY START
            10 ;THIS ROUTINE PRINTS A LINE TO THE PRINTER BY CALLING SIO E459
            20 SIOV       =      $E459   SIO VECTOR
009B        30 CR         =      $9B     EOL
0040        40 PRNTID     =      $40     PRINTER SERIAL BUS ID
004E        45 MODE       =      $4E     NORMAL MODE
001C        50 PTIMOT     =      $001C   TIMEOUT LOCATION
0300        60 DDEVIC     =      $300    DEVICE SERIAL BUS ID
0301        70 DUNIT      =      $301    SERIAL UNIT NUMBER
0302        80 DCOMND     =      $302    SIO COMMAND
0303        90 DSTATS     =      $303    SIO DATA DIRECTION
0304        0100 DBUFLO   =      $304    BUFFER LOW ADDRESS
0305        0110 DBUFHI   =      $305    BUFFER HIGH ADDRESS
0306        0120 DTIMLO   =      $306    SIO TIMEOUT
0307        0130 DTIMHI   =      $307
0308        0140 DBYTLO   =      $308    BUFFER LENGTH
0309        0150 DBYTHI   =      $309
030A        0160 DAUX1    =      $30A    AUXILARY BYTE---PRINTER MODE
030B        0170 DAUX2    =      $30B    AUXILARY BYTE---NOT USED
            0180 ;
3000 455841 0190 MESS     .BYTE  "EXAMPLE 12",CR
3001 4D504C
3005 452031
3009 329B
            0200 ;
300B A940   0220          LDA    #PRNTID     SET BUS ID
300D 8D0003 0230          STA    DDEVIC
3010 A901   0240          LDA    #1          SET UNIT NUMBER
3012 8D0103 0250          STA    DUNIT
3015 A94E   0260          LDA    #MODE
3017 8D0A03 0270          STA    DAUX1       PRINTER MODE NORMAL
301A A901   0275          LDA    #1
301C 8D0B03 0280          STA    DAUX2       UNUSED
301F 8D0703 0290          STA    DTIMHI      TIMEOUT<256 SECS
3022 A51C   0300          LDA    PTIMOT      SET SIO TIMEOUT FOR PRINTER
3024 8D0603 0310          STA    DTIMLO
3027 A900   0320          LDA    #MESS&255
3029 8D0403 0330          STA    DBUFLO      SET MESS AS BUFFER
302C A930   0340          LDA    #MESS/256
302E 8D0503 0350          STA    DBUFHI
3031 A980   0360          LDA    #$80        SET SIO DATA DIRECTION FOR
3033 8D0303 0370          STA    DSTATS      PERIPHERAL TO RECEIVE
3036 A957   0380          LDA    #'W         SIO COMMAND WRITE
3038 8D0203 0390          STA    DCOMND
303B 2059E4 0410          JSR    SIOV        CALL SIO
303E 3001   0420          BMI    ERROR
3040 00     0430 GOOD     BRK
3041 00     0440 ERROR    BRK
```

Figure 8-15 SIO Call to Dump Line to Printer

SIO Interrupts

SIO uses three IRQ Interrupts to control serial bus communications with serial bus devices:

IRQ	Location, Length	Function
VSERIR	[$020A,2]	SERIAL INPUT READY
VSEROR	[$020C,2]	SERIAL OUTPUT NEEDED
VSEROC	[$020E,2]	TRANSMISSION FINISHED

All program execution is halted while SIO uses the serial bus for communication. Even though nothing else is happening during a serial bus transfer, the actual I/O itself is interrupt driven. The method of communicating between SIO and the interrupt handlers is known as the semaphore method. The mainline code waits in a loop until the interrupt handlers signal it that they are finished. For instance, during output, SIO places a byte to be transferred into the serial output shift register located in POKEY. It then enters a loop and watches a flag which will be set when the requested I/O is completed. During this time POKEY is sending the bits out over the serial line.

When the byte has been sent, a Serial Output Needed IRQ is generated. This IRQ then vectors to a routine which loads the next byte in the buffer into the serial output register. This process continues until the entire buffer has been sent. After taking care of checksum values, the interrupt handler then sets the transmit done flag. Meanwhile, SIO has been patiently looping, watching for this flag to be set. When it sees that the flag has been set, SIO exits back to the calling routine.

The SIO execution for input is similar. POKEY generates an IRQ (VSERIR) to inform SIO a byte has been received in the serial input shift register (SERIN). The interrupt handler for VSERIR then stores the byte in a buffer and checks whether the end of the buffer has been reached. If so, it sets the transmit done flag.

You may have noticed from the above explanation that SIO wastes some time idling while waiting for POKEY to send or receive information on the bus. Because the vectors for the three SIO IRQ service routines are RAM vectors, they can be used by your own handlers to improve system I/O performance. Indeed, this is how the ATARI 850 Interface Module is able to do concurrent I/O. That handler takes over the SIO IRQ vectors and points them to the module's own IRQ routines while in concurrent I/O. This allows the calling program to continue to execute while the interface Module sends commands and data over the serial line.

Using CIO from BASIC

Most of the CIO functions (OPEN, CLOSE, etc.) are available through calls from BASIC using the OPEN, GET and PUT statements. However, BASIC lacks one set of the functions of CIO, the ability to do non-record I/O more than a byte at a time (GETCHRS and PUTCHRS).

The ability to input or output a buffer of characters is a powerful asset. An assembly language routine can be loaded directly into memory from a disk file for instance. Or a high resolution graphics image can be loaded directly into the screen memory area. A common method of improving a BASIC language program's performance is to provide a machine language program to handle certain functions that BASIC executes slowly. Unfortunately, finding a place in RAM for such a routine can be a knotty problem. One solution is to place the routine into an area reserved for a string, and calling the routine with a USR call to ADR(string). Since the address of a BASIC string may shift during program editing, the assembly language routine must be relocatable. Therefore, unmodified memory reference instructions to addresses within the string will not work.

The subroutine in Figure 8-16 avoids the use of strings by loading a routine into Page 6 RAM. Thus the assembly language routine need not be relocatable. Control data is POKEd into an IOCB to read an assembly language routine directly into RAM at the address it was assembled. The BASIC subroutine in Figure 8-16 can also be used to output data directly from memory with the user specifying both the location and the length of the data buffer.

```
30 REM THIS PROGRAM LOADS PAGE 6 FROM THE FILE D:TEST
100 DIM FILE$(20),CIO$(7):CIO$="hhh*LVd"
106 REM CIO$ IS PLA,PLA,PLA,TAX,JMP $E456 (CIOV)
110 FILE$="D:TEST":REM - *FILE NAME
120 CMD=7:STADR=1536:GOSUB 30000
130 IF ERROR=1 THEN ? "TRANSFER COMPLETE":STOP
135 ? "ERROR # ";ERROR;" OCCURRED AT LINE # ";PEEK(186)+256*PEEK(187)
200 END
300 REM _ CIO SETUP SUBROUTINE
310 REM
30000 REM
30001 REM
30002 REM THIS ROUTINE LOADS OR SAVES MEMORY FILE FROM BASIC
30003 REM BY SETTING UP AN IOCB AND CALLING CIO DIRECTLY
30004 REM
30006 REM ON ENTRY CMD=7 MEANS LOAD MEMORY
30008 REM _        CMD=11 MEANS SAVE MEMORY
30009 REM _        STADR= THE ADDRESS TO LOAD OR SAVE MEMORY
30010 REM _        BYTES= THE NUMBER OF BYTES TO SAVE OR LOAD
30011 REM _        IOCB= THE IOCB TO USE
30012 REM _        FILE$= DESTINATION FILE NAME
30013 REM _
30014 REM ON EXIT ERROR=1 MEANS SUCCESSFUL COMMAND
30018 REM _       ERROR<>1 THEN ITS AN ERROR STATUS
30019 REM
30020 REM _ *** IOCB EQUATES ***
```

```
30022 REM
30024 IOCBX=IOCB*16:ICCOM=834+IOCBX:ICSTA=835+IOCBX
30026 ICBAL=836+IOCBX:ICBAH=837+IOCBX
30028 ICBLL=840+IOCBX:ICBLH=841+IOCBX
30029 REM
30030 AUX1=4:IF CMD=11 THEN AUX1=8
30035 TRAP 30900:OPEN #IOCB,AUX1,0,FILE$
30040 TEMP=STADR:GOSUB 30500
30090 POKE ICBAL,LOW:POKE ICBAH,HIGH
30100 TEMP=BYTES:GOSUB 30500
30130 POKE ICBLL,LOW:POKE ICBLH,HIGH
30140 POKE ICCOM,CMD:ERROR=USR(ADR(CIO$),IOCBX)
30150 ERROR=PEEK(ICSTA):RETURN
30200 REM
30300 REM _ ***ROUTINE RETURNS HIGH,LOW BYTE OF 16 BIT NUMBER
30400 REM
30500 HIGH=INT(TEMP/256):LOW=INT(TEMP-HIGH*256):RETURN
30550 REM
30600 REM ***TRAP HERE IF ERROR OCCURS DURING ROUTINE****
30900 ERROR=PEEK(195) 30920 CLOSE #IOCB:RETURN
```

Figure 8-16 BASIC Direct CIO Call

REAL TIME PROGRAMMING

Most of the time in programming we have the luxury of ignoring time considerations. Usually we don't care how long a program takes to run, or don't bother to measure precise timing values on subroutine execution. Sometimes, however, timing considerations play an important role in the performance of the program, and then we enter the world of Real Time Programming. Such cases arise often with the ATARI Home Computer System. Much more so than with most other small computers, this system thrives on real time control. The very time base upon which the internal circuitry operates was precisely chosen so that the computer hardware would be in complete synchrony with a specific signal - namely, the television signal.

In order to get clean, crisp graphics and special effects, the hardware in the ATARI Home Computer System is slaved to the local television signal. Unfortunately, there are several "standard" television signals in use in various countries. In the United States, the standard is the NTSC system: 60 frames per second, 262 horizontal lines per frame, and 228 color clocks per line. The 262 lines comes about only because the ATARI Home Computer System generates a non-interlaced signal; the real standard calls for 525 lines with half being shown each frame. Some European countries use a standard called PAL: 50 frames per second, 312 lines per frame. The result is that timing considerations are different on NTSC vs. PAL systems. Refer to Section 2, ANTIC and the Display List, for a more detailed discussion of the television signal. The remarks in this subsection are based on NTSC systems.

Synching to the Television Signal

The 6502 processor is synchronized to the television signal in two ways, a coarse and a fine synchronization. Coarse synchronization is derived by having the same signal that synchronizes television receivers to the transmitted television signal cause a system interrupt. This signal is called vertical blank, and in television sets it is the cue to turn off the electron beam and begin retracing to the top of the screen in preparation for another frame. This same signal is presented to the computer as a nonmaskable interrupt. To the programmer, this provides a regularly occurring interrupt that can be used for everything from sound and timing information to a crude multiprogramming method. For a detailed discussion of Vertical Blank processing, see the subsection on interrupt Processing Structure.

An even finer correlation between the 6502 processing and the television signal was achieved by setting the system clock rate to 1.79 MHz. This results in a direct relationship between the time it takes to execute a machine instruction and the distance the electron beam travels on the screen. For instance, during the time it takes to execute the shortest 6502 instruction (2 cycles), the beam moves four color clocks, or one OS mode 0 character width across the screen. This precise correlation of timing allows a skilful programmer to produce graphic effects in the middle of a single scan line. A note of caution is in order, however. ANTIC's direct memory access makes that this intraline timing very uncertain, and will vary depending upon the mode selected and other factors. In actual usage, this means that each intraline change must be dealt with and tested as a special case.

Hardware Timers

There are four countdown timers built into the POKEY chip. They function as reusable "hardware subroutines". The most common use of these timers is in conjunction with the audio channels for producing sound effects (see Section 7, Sound). They may also be used as straight countdown timers, since they generate an IRQ interrupt (see subsection on Interrupt Processing Structure). Each timer is associated with a frequency register which holds the initial value for the timer. When the hardware register STIMER is written to, this initial value is loaded and the countdown timer started. When the timer counts down to zero, an IRQ Interrupt request is generated. It is important to note that only timers 1, 2 and 4 have interrupt vectors for processing. The following steps must be taken to activate any of the timers.

Step 1. Set AUDCTL [$D208] with proper value to select clocking frequency for the desired timer.
Step 2. Set the volume output to 0 for the audio channel associated with the selected timer (AUDC1, AUDC2, AUDC4 [$D201, $D203, $D207]).
Step 3. Set AUDF1 , AUDF2 or AUDF4 [$D200, $D202, $D206] with the desired number of clock intervals to count.
Step 4. Set up routine to process timer interrupt.
Step 5. Change timer interrupt vector to point to processing code (VTIMR1, VTIMR2, VTIMR4 [$0210, $0212, $0214]). *** Note. Due to a bug in the source code, the original version of the operating system will never vector through VTIMR4. This has been fixed in Rev. B.
Step 6. Set bits 0, 1 or 2 in IRQEN and the OS shadow POKMSK [$D20E and $0010] to enable interrupts from timers 1, 2 or 4.
Step 7. Write any value to register STIMER [$D209] to load and start the count down timers.

One complication of working with these timers is that the 6502's response to them will be pre-empted and possibly delayed by ANTIC's direct memory access, display list interrupts or vertical blank processing.

Software Timers

There are 6 system software timers:

Name	Location	Vector or Flag
RTCLOK	[$0012,3]	none
CDTMV1	[$0218,2]	CDTMA1 [$0226,2]
CDTMV2	[$021A,2]	CDTMA2 [$0228,2]
CDTMV3	[$021C,2]	CDTMF3 [$022A,1]
CDTMV4	[$021E,2]	CDTMF4 [$022C,1]
CDTMV5	[$0220,2]	CDTMF5 [$022E,1]

All of the system timers are decremented as part of the vertical blank (VBLANK) process. If the VBLANK process is disabled or intercepted, the timers will not be updated.

The real time clock (RTCLOK) and system timer 1 (CDTMV1) are updated during immediate VBLANK, Stage 1. RTCLOK counts up from 0 and is a three-byte value. When RTCLOK reaches its maximum value (16,777,216) it will be reset to zero. RTCLOK can be used as a real. time clock as Figure 8-17 shows.

Because the system timers are updated as part of the VBLANK process, special care is needed to set them correctly. A system routine called SETVBV [$E45C] is used to set them. The call to SETVBV is:

REG :
X contains the timer value high byte
Y contains the timer value low byte
A contains the timer number 1-5

Example:

```
LDA #1      ;Set system timer 1
LDY #0
LDX #2      ;value is $200 VBLANK periods
JSR SETVBV  ;Call system routine to set timer
```

System timers 1-5 are 2-byte counters. They may be set to a value using the SETVBV routine. The OS then decrements them during VBLANK. Timer 1 is decremented during immediate VBLANK, Stage 1. Timers 2-5 are decremented during immediate VBLANK, Stage 2. Different actions are taken by the OS when the different timers are decremented to 0.

System timers 1 and 2 have vectors associated with them. When timer 1 or 2 reaches 0, the OS simulates a JSR through the vector for the given timer. Figure 8-7 gives the vectors for the two timers.

System timers 3-5 have flags that are normally SET (i.e. non-zero). When either of the three timers count to 0, the OS will clear (zero) that timer's flag. You may then test the flag and take appropriate action.

Timers 1-5 are general purpose software timers that may be used for a variety of applications. For example, timer 1 is used by SIO to time serial bus operations. If the timer counts to zero before an bus operation is complete, a "timeout" error is returned. Timer 1 is set to various values depending on the device being accessed. This ensures that, while a device has ample time to answer an I/O request, the computer will not wait indefinitely for a non-existent device to respond. The cassette handler uses timer 3 to set the length of time to read and write tape headers. Figure 8-18 shows an example using timer 2 to time a sound acting as a metronome.

The software timers are generally used when the time scale involved is greater than one VBLANK period. For time durations shorter than this, either the hardware timers or some other method must be employed.

```
1 POKE 752,1
3 ? "+":REM CLEAR SCREEN (+=ESC-CTRL-CLR)
4 ? "HOUR";:Input HOUR:? "MINUTE";:Input MIN:? "SECOND";:Input SEC
5 CMD=1:GoSub 45
6 ? "+";HOUR;" : ";MIN;" : ";SEC:? " ": ? " "
7 CMD=2:GoSub 45
9 ? "";HOUR;":";MIN;":";SEC;"   ":GoTo 7
10 REM THIS IS A DEMO OF THE REAL TIME CLOCK
20 REM THIS ROUTINE ACCEPTS AN INITIAL TIME In HOURS,MINUTES, And SECONDS
30 REM IT SETS THE REAL TIME CLOCK To ZERO
40 REM THE CURRENT VALUE OF RTCLOCK IS USED To ADD THE INITIAL TIME To Get
42 REM THE CURRENT TIME HOUR,MIN,SEC
45 HIGH=1536:MED=1537:LOW=1538
50 REM
60 REM ******ENTRY POINT****** 65 REM
70 On CMD GoTo 100,200
95 REM
96 REM ****INITIALIZE CLOCK*****
97 REM
100 POKE 20,0:POKE 19,0:POKE 18,0
105 Dim CLOCK$(50)
106 CLOCK$=" "
107 GoSub 300
110 IHOUR-HOUR:IMIN=MIN:ISEC=SEC:Return
197 REM
198 REM *******Read CLOCK*****
199 REM
200 REM
201 A=USR(ADR(CLOCK$))
210 TIME-C(((PEEK(HIGH)*256)+PEEK(MED))*256)+PEEK(LOW))/59.923334
220 HOUR=INT(TIME/3600):TIME=TIME-(HOUR*3600)
230 MIN=INT(TIME/60):SEC=INT(TIME-(MIN*60))
235 SEC=SEC+ISEC:If SEC>60 Then SEC=SEC-60:MIN=MIN+1
236 MIN=MIN+IMIN:If MIN>60 Then MIN=MIN-60:HOUR=HOUR+1
237 HOUR=HOUR+IHOUR
240 HOUR=HOUR-(INT(HOUR/24))*24
250 Return
300 For J=l To 38:Read Z:CLOCK$(J,J)=CHR$(Z):Next J:Return
310 DATA 104,165,18,141,0,6,165,19,141,1,6,165
320 DATA 20,141,2,6,165,18,205,0,6,208,234
330 DATA 165,19,205,1,6,208,227,165,20,205,2,6,208,220,96
```

Figure 8-17 Real Time Clock

```
1 REM THIS IS A BASIC PROGRAM TO CONTROL THE RATE OF A METRONOME
2 REM
3 REM
5 PRINT "+":REM                       CLEAR SCREEN
10 X=10:REM                           INITIAL VALUE FOR RATE
20 FOR J=1 TO 10:NEXT J:REM           SOFTWARE DELAY LOOP
50 IF STICK(0)=14 THEN X=X+1 :REM     STICK FORWARD MEANS SPEED UP RATE
51 IF STICK(0)=13 THEN X=X-1 :REM     STICK BACK MEANS SLOW METRONOME RATE
52 IF X<1 THEN X=1:REM                NEVER GO BELOW ONE
53 IF X>255 THEN X=255:REM            OR ABOVE 255
54 REM                                PRINT BEATS/MINUTE
56 ? "";INT(3600/X);" BEATS/MINUTES       "
60 POKE 0,X:REM                       LOCATION $0000 HOLDS THE RATE FOR
70 NEXT I :REM                        THE FOLLOWING ASSEMBLY ROUTINE
```

Figure 8-18 BASIC language Metronome Routine

```
40              *=$600
50 ;METRONOME   ROUTINE...USES $0000 TO PASS THE METRONOME RATE
60 ;
70 AUDF1        =       $D200   AUDIO FREQUENCY REGISTER
80 AUDC1        =       $D201   AUDIO CONTROL REGISTER
90 FREQ         =       $08     AUDF1 VALUE
0100 VOLUME     =       $AF     AUDC1 VALUE
0110 OFF        =       $A0     TURN OFF VOLUME
0120 SETVBV     =       $E45C   SET TIMER VALUE ROUTINE
0130 XITVBV     =       $E462
0140 CDTMV2     =       $021A   TIMER 2
0150 CDTMA2     =       $0228   TIMER 2 VECTOR
0160 ZTIMER     =       $0000   ZPAGE VBLANK TIMER VALUE
0170 ;
0180 START      LDA     #10
0190            STA     ZTIMER
0200 ;      SET THE TIMER VECTOR
0220 ;
0230 INIT       LDA     #CNTINT&255
0240            STA     CDTMA2
0250            LDA     #CNTINT/256
0260            STA     CDTMA2+1
0270 ;
0280 ;      SET THE TIMER VALUE AFTER THE VECTOR
0290 ;
0300            LDY     ZTIMER   SET TIMER TWO TO COUNT
0310            JSR     SETIME
0320            RTS
0340 ;      METRONOME COUNT DOWN VECTORS TO HERE
0380 ;      SET UP AUDIO CHANNEL FOR MET CLICK
0390
0400 CNTINT     LDA     #VOLUME
0410            STA     AUDC1
0420            LDA     #FREQ
0430            STA     AUDF1
0435            LDY     #$FF  DELAY
0440 DELAY      DEY
0442            BNE     DELAY
0450            STY     AUDC1
0460            JMP     INIT
0480 ;
0490 ;      SUBROUTINE TO SET TIMER
0500 ;
0520 SETIME     LDX     #0              NO TIME >256 VBLANKS
0530            LDA     #2              SET TIMER 2
0540            JSR     SETVBV          SYSTEM ROUTINE TO SET TIMER
0550            RTS
0560            *=$2E2
0570            .WORD   START
0580            .END
```

Figure 8-19 Assembly language Metronome Routine

FLOATING POINT PACKAGE

The Floating Point Package (FPP) is a set of integrated routines that provide an extended arithmetic capability for the OS. These routines are combined in a separate ROM chip that is provided as part of the ATARI 10K OPERATING SYSTEM. The FPP is located at hexadecimal

addresses $D800 - $DFFF. It has not been changed in the Revision B version of the OS. The following paragraphs detail the internal representation of numbers, the actual routines available for use, and their proper calling sequence. An assembly language program example is included to illustrate how to access the FPP from user programs.

Internal Representation

The FPP configures numbers internally as 6-byte quantities. Each number consists of a 1-byte exponent and a 5-byte mantissa in Binary Coded Decimal (BCD) format. This representation was chosen to minimize rounding errors that might occur in some math routines.

The sign bit of the exponent byte provides the sign of the mantissa, 0 for positive, 1 for negative. The least significant 7 bits of the exponent byte provide the exponent as a power of 100 in "excess 64 notation". In excess 64 notation, the value 64 is added to the exponent value before it is placed in the exponent byte. This allows the full range of exponents, positive and negative, to be expressed without having to use the sign bit.

The mantissa is always normalized such that the most significant byte is non-zero. However, since the mantissa is in BCD format, and the exponent represents powers of 100 and not 10, either 9 or 10 digits of precision may result. There is an implied decimal point to the right of the first mantissa byte so that an exponent that is less than 64 (40 hex) indicates a number less than 1.

EXAMPLES(Format values are in hex)

Number: 0.02 2 * 100**-l
Format: 3F 02 00 00 00 00 (exponent= 40-1)

Number: -0.02 -2 * 100**-l
Format: BF 02 00 00 00 00 (exponent= 80+40-1)

Number: 37.0 = 37 * 100**0
Format: 40 37 00 00 00 00 (exponent= 40+0)

Number: -460312 = -46.0312 * 100**2
Format: C2 46 03 12 00 00 (exponent= 80+40+2)

The number zero is handled as a special case, and is represented as a zero exponent and zero mantissa. Either the exponent or the first mantissa byte may be tested for zero.

The dynamic range of numbers that can be represented by this scheme is 10**-98 to 10**+98.

Memory Utilization

Two areas of RAM memory are used in implementing the FPP. They are:

```
$00D4 - $00FF in Page zero
$057E - $05FF in Page five
```

These areas are used both for control parameter storage and to simulate several floating point registers. The two pseudo-registers of primary interest are called FRO and FR1 (locations $00D4-$00D9 and $00EO- $00E5 respectively). Each of these pseudo-registers is six bytes in length and will hold a number in floating point representation. A two- byte pointer is used in pointing to a floating point number. This is called FLPTR and resides at $00FC.

Buffer areas must be provided for text strings in converting between floating point numbers and ATASCII text strings. The output buffer is called LBUFF, a 128 byte block from $0580 to $05FF. The input buffer is set by the two byte pointer INBUFF at $00F3. Also, a one byte index CIX at $00F2 is used as an offset into the buffer pointed to by INBUFF.

A typical sequence to use the floating point package from an assembly language routine would be as follows. First an ATASCII string that represents one of the numbers to be used by a math routine would be stored in a buffer anywhere in memory. Next, pointer INBUFF would be set to

point to the beginning of this string. Also, the index value, CIX, should be set to 0. The number is then ready to be converted to a floating point representation, so routine AFP would be called. This would result in a floating point number being placed in FRO where it could be used in any of the FPP operations. After the conclusion of the mathematical operations, the floating result would be in FR0. Calling the routine FASC would convert this number to an ATASCII string located in LBUFF. Refer to Figure 8-21 for an example of this process.

To use 16-bit values with the FPP, place the two bytes of the number into the lowest two bytes of FRO ($D4 and $D5) and JSR IFP, which converts the integer to its floating point representation., leaving the result in FRO. Subroutine FP1 performs the reverse operation.

The chart on the next page lists the functions available, their ROM addresses, pseudo registers affected and approximate maximum computation time.

FLOATING POINT ROUTINES

Name	**Address**	**Function**	**Result**	**MAX. TIME (microsec.)**
AFP	D800	ATASCII to floating point	FR0	3500
FASC	D8E6	Floating point to ATASCII	LBUFF	950
IFP	D9AA	Integer to floating point	FR0	1330
FPI	D9D2	Floating point to Integer	FR0	2400
FSUB	DA60	FR0-FR1 Subtraction	FR0	740
FADD	DA66	FR0+FR1 Addition	FR0	710
FMUL	DADB	FR0*FR1 Multiplication	FR0	12000
FDIV	DB28	FR0/FR1 Division	FR0	10000
FLDOR	DD89	Floating Load using X,Y	FR0	70
FLDOP	DD8D	Floading Load using FLPTR	FR0	60
FLD1R	DD98	Floating Load using X,Y	FR1	70
FLD1P	DD9C	Floating Load using FLPTR	FR1	60
FSTOR	DDA7	Floating store using X,Y	FR0	70
FSTOP	DDA8	Floating store with FLPTR	FR0	70
FMOVE	DDB6	Move FR0	FR1	60
PLYEVL	DD40	Polynomial evaluation	FR0	88300
EXP	DDC0	Exponentiation - e**FR0	FR0	115900
EXP1O	DDCC	Exponentiation - 10**FR0	FR0	108800
LOG	DECD	Natural logarithm	FR0	136000
LOG10	DED1	Base 10 logarithm	FR0	125400
ZFR0	DA44	Set to zero	FR0	80
AF1	DA46	Set register in X to zero	varies	80

Figure 8-20 Floating Point Routines

```
0000        20         *= $4000  ; ARBITRARY STARTING POINT
DDB6        30   FMOVE  = $DDB6
DA60        40   FSUB   = $DA60
0482        50   FTEMP  = $0482
```

```
DDA7         60   FSTOR  = $DDA7
D8E6         70   FASC   = $D8E6
00F3         80   INBUFF = $00F3
D800         85   AFP    = $D800
00F2         90   CIX    = $00F2
0580         100  LBUFF  = $0580
009B         120  CR     = $9B
0009         130  PUTREC = $09
0005         140  GETREC = $05
E456         150  CIOV   = $E456
0342         160  ICCOM  = $0342
0344         170  ICBAL  = $0344
0348         180  ICBLL  = $0348
             190  ;
             200  ; FLOATING POINT DEMONSTRATION
             210  ; READS TWO NUMBERS FROM SCREEN EDITOR,
             215  ; CONVERTS THEM TO FLOATING POINT,
             220  ; SUBTRACTS THE FIRST FROM THE SECOND,
             225  ; STORES THE RESULT IN FTEMP,
             230  ; WHICH IS A USER DEFINED FP REGISTER,
             240  ; AND DISPLAYS THE RESULT.
4000 205340  260  START   JSR GETNUM      ; GET 1ST NUMBER FROM E:
4003 20B6DD  270          JSR FMOVE       ; MOVE NUMBER FROM FRO TO FR1
4006 205340  280          JSR GETNUM      ; GET 2ND NUMBER FROM E:
4009 2060DA  290          JSR FSUB        ; FRO <-- FR0-FR1
400C 900A    300          BCC NOERR       ; SKIP IF NO ERROR
400E A981    340          LDA #ERRMSG&255 ; IF ERR., DISPLAY MESSAGE
4010 8D4403  350          STA ICBAL
4013 A940    360          LDA #ERRMSG/256
4015 4C3940  370          JMP CONTIN
4018 A282    390  NOERR   LDX #FTEMP&255  ; STORE RESULT IN FTEMP
401A A004    400          LDY #FTEMP/256
401C 20A7DD  410          JSR FSTOR
             420  ;
             430  ; CONVERT NUMBER TO ATACSII STRING.
             440  ; FIND END OF STRING,
             445  ; CHANGE NEGATIVE NUM. TO POSITIVE,
             450  ; AND ADD CARRIAGE RETURN.
401F 20E6D8  470          JSR FASC        ; FP TO ATASCII, RESULT IN LBUFF
4022 A0FF    480          LDY #$FF
4024 C8      490  MLOOP   INY
4025 B1F3    500          LDA (INBUFF),Y  ; LOAD NEXT BYTE
4027 10FB    510          BPL MLOOP       ; IF POSITIVE, CONTINUE
4029 297F    520          AND #$7F        ; IF NOT, MASK OFF MSBIT
402B 9lF3    530          STA (INBUFF),Y
402D C8      540          INY
402E A99B    550          LDA #CR         ; STORE CARRIAGE RETURN
4030 9lF3    560          STA (INBUFF),Y
             570  ;
             580  ; DISPLAY RESULT
4032 A5F3    600          LDA INBUFF      ; GET BUFFER ADDRESS
4034 8D4403  610          STA ICBAL
4037 A5F4    620          LDA INBUFF+1
4039 8D4503  630  CONTIN  STA ICBAL+1
403C A909    640          LDA #PUTREC     ; COMMAND FOR PUT RECORD
403E 8D4203  650          STA ICCOM
4041 A928    660          LDA #40         ; SET BUFFER LENGTH = 40
4043 8D4803  670          STA ICBLL
4046 A900    690          LDA #0
4048 8D4903  700          STA ICBLL+1
404B A200    710          LDX #0          ; IOCB 0 FOR SCREEN EDITOR
404D 2056E4  720          JSR CIOV        ; CALL CIO
4050 4C0040  730          JMP START       ; DO IT AGAIN
             740
             750  ; GET ATASCII STRING FROM E:
             755  ; CONVERT TO FP, RESULT IN FR0
4053 A905    780  GETNUM  LDA #GETREC     ; GET RECORD (ENDS WITH CR)
4055 8D4203  790          STA ICCOM
4058 A980    800          LDA #LBUFF&255  ; SET BUFFER ADDRESS = LBUFF
405A 8D4403  810          STA ICBAL
405D A905    820          LDA #LBUFF/256
405F 8D4503  830          STA ICBAL+1
4062 A928    840          LDA #40         ; SET BUFFER LENGTH =40
4064 8D4803  850          STA ICBLL
4067 A900    860          LDA #0
4069 8D4903  870          STA ICBLL+1
406C A200    880          LDX #0          ; IOCB 0 FOR SCREEN EDITOR
406E 2056E4  890          JSR CIOV        ; CALL CIO
4071 A980    900          LDA #LBUFF&255  ; STORE BUFFER ADD. IN INBUFF
4073 85F3    910          STA INBUFF
```

```
4075 A905   920         LDA #LBUFF/256
4077 85F4   930         STA INBUFF+1
4079 A900   940         LDA #0          ; SET BUFFER INDEX = 0
407B 85F2   950         STA CIX
407D 2000D8 960         JSR AFP         ; CALL ATASCII TO FP
4080 60     970         RTS
4081 45     980  ERRMSG  .BYTE "ERROR",CR
4082 52
4083 52
4084 4F
4085 52
4086 9B
            1000 ; ROUTINE START INFO
4087        1020        * = $2E0
02E0 0040   1030        .WORD START
02E2        1040        .END
```

Figure 8-21 Floating Point Example

Chapter IX

The Disk Operating System

9 The Disk Operating System

The Disk Operating System (DOS) is an extension of the OS that allows you to access the ATARI 810' Disk Drive mass storage device as you would access any of the other input/out devices. DOS has three primary components, the resident disk handier, the File Management System (FMS), and the Disk Utility Package (DUP). The resident disk handler is the only part of DOS provided in the OS ROM. FMS and DUP both reside on diskette, with FMS being loaded ("booted") into memory at power-up. DUP is not automatically loaded at power up, but requires an explicit request from an application program. If the BASIC language cartridge is present, for instance, DUP is not loaded until a DOS statement is issued. The following subsections describe each of these components in more detail and present other related topics necessary for effective utilization of DOS. You should note that the comments in this section refer to version 2.0S of the Disk Operating System, which is substantially different from earlier versions of DOS. For complete technical details regarding DOS, you should also refer to the OPERATING SYSTEM Manual, and the DISK OPERATING SYSTEM II Manual.

The Resident Disk Handler
The resident disk handler is the simplest part of DOS. The disk handier does not conform to the normal CIO calling sequence as observed by other Device Handlers (see section 8, the Operating System, for details on using the Centralized Input/Output Subsystem). In DOS 2.0S, the resident disk handler is used only during the initial boot process. From then on, additional code in the FMS is used in accessing the disk drives. The relationship of the disk handier to the I/O subsystem is shown in Figure 8-8 of this manual.

The Device Control Block (DCB) is used to communicate with the disk handler. Figure 8-10 illustrates the structure of the DCB. The calling sequence for the disk handler is:

```
             ;caller has already set up DCB
JSR DSKINV   ;system routine Vector to the resident Disk Handler
BPL OKAY     ;Branch if success, Y Reg. = 1
             ;Else y reg. = error status (DCBSTA also has error)
```

The disk handler is a subroutine that is used to support the physical transfers of data between the 6502 inside the ATARI Home Computer System, and another processor located inside the ATARI 810 Disk Drive. This data transfer occurs over the serial input/output bus. The OS resident disk handier supports four functions:

```
FORMAT                  Issue a Format command to the disk controller
READ SECTOR             Read a specified sector
WRITE/VERIFY SECTOR     Write sector; check sector to see if written
STATUS                  Ask the disk controller for its status
```

The FORMAT command clears all the tracks on the diskette and writes sector addresses onto the tracks. No file structure is put on the diskette by this command. The data portion of each sector is set to all zeros, and the initial Volume Table of Contents and the File Directory are established. For more information on the physical layout of data on a diskette, refer to the OPERATING SYSTEM Manual and the subsection on FMS Disk Utilization.

You should note that all I/O from the disk handier is sector-oriented. The sector I/O commands can be used to read and write any sector on the diskette. You can use them to implement your own file structure. Section 10 of the OPERATING SYSTEM Manual has an example of using the disk handler to write a boot tile.

The STATUS function is used to determine the status of the disk drive. This command causes the disk drive to transmit four bytes that define its current status. These bytes are loaded into DVSTAT [$02EA,4]. The first byte is a command status byte and contains the following status bits:

Bit-0 = 1 indicates an invalid command frame was received.
Bit-1 = 1 indicates an invalid data frame was received.
Bit-2 = 1 indicates that a PUT operation was unsuccessful.
Bit-3 = 1 indicates that the disk is write protected.
Bit-4 = 1 indicates active/standby.

The second byte is a hardware status byte and contains a copy of the status register of the INS1771-1 Floppy Disk Controller chip used in the disk controller. The third byte is a timeout byte that contains a controller- provided maximum timeout value (in seconds) to be used by the handler. The fourth byte is unused. You can use the STATUS command for several purposes. Since the device timeout value for a STATUS command is less than that for the other commands, you can use it to see if a specific disk drive is connected. If the disk handler returns a device timeout error, you know the disk drive is not connected.
File Management System
The File Management System (FMS) is a nonresident device handler that uses the normal device handler-CIO interface. FMS is not present in the OS ROM. It is booted in at power-up if a diskette containing DOS is present.

FMS, like the other device handlers, gets I/O control data from CIO. FMS then uses its own disk handler to input and output to the diskette. The additional disk handler code was provided primarily to overcome an operating system bug. This bug is the result of an incorrect 16-bit compare of buffer pointers that may occur during SIO transfers. Specifically, it occurs when a buffer ends on a page boundary. However, since the result of this patch is to place a disk handler in RAM, it is possible to customize DOS somewhat. The hardware in the disk drive itself is capable of another function not supported by the resident disk handler. This function is a WRITE SECTOR WITHOUT VERIFY command. Even though some reliability is sacrificed, disk writes occur faster. To perform this customization from BASIC, you need to type:

```
POKE 1913,80
```

for fast Write (Write without Read Verify). If you want to restore the Write with Read verify, type:

```
POKE 1913,87
```

FMS is called by setting up an IOCB and calling CIO. FMS supports some special CIO functions not available to other handlers:

`FORMAT`	FMS calls the disk handler to format the diskette. After a successful format, FMS writes some file structure data on the diskette.
`NOTE`	FMS returns the current value of the file pointer.
`POINT`	FMS sets the file pointer to a specified value.

The subsection on Random Access contains instructions on using NOTE and POINT.

Disk I/O

You can access all the standard file I/O calls through CIO. In BASIC this means using the I/O commands, such as OPEN, CLOSE, GET, PUT and XIO. In assembly language you have to set up the IOCB yourself and call CIO.

To do any disk I/O, you must first OPEN a file. The BASIC syntax for the OPEN command is:

```
OPEN #IOCB,ICAX1,0,"D:MYPROG.BAS"
```

The # IOCB selects one of the eight IOCBs available (see the CIO subsection In section 8, the Operating System, for IOCB setup). You should avoid IOCBs #0, 6, and 7 since they are used by the OS and BASIC at various times. ICAX1 is the OPEN type code. The bits for the type code are:

BIT	7	6	5	4	3	2	1	0
	x	x	x	x	W	R	D	A

Where: A is Append
D is Directory
R is Read
W is Write
x is unused

Figure 9-1 Auxiliary Byte Layout for the OPEN Command

The various values for ICAX1 are discussed in Section 5 of the OPERATING SYSTEM Manual. Some of the key things to note about the various OPEN modes are:

ICAX1=6 This is used to OPEN the diskette directory. Records READ are the diskette directory entries.
ICAX1=4 READ mode.
ICAX1=8 WRITE mode. Any existing file opened in this mode is first deleted. The first bytes written will be at the start of the file.
ICAX1=9 WRITE APPEND mode. The file is left intact. Bytes written to this file are put at the end of the file.
ICAX1=12 UPDATE mode. This mode allows both READ and WRITE to the file. Bytes read and written start at the first byte in the file.
ICAX1=13 Not supported.

There are two types of I/O you can use to transfer data between your program and the disk, record or character.

Character I/O means that the data In a file is a sequential string of bytes. DOS interprets this list of bytes as data, with none of the values being interpreted as control characters. An example of character data (all values are in Hex):

```
00 23 4F 55 FF 34 21.
```

Record I/O means that data in a file is made up of a set of records. A record is a group of bytes followed by an End of Line (EOL) character with the value $9B. An example of two records is:

```
00 23 4F 55 FF 34 9B 21 34 44 9B
|   record 1        | record 2 |
```

Record and character I/O to files can be done in any arbitrary order. Indeed, data created as records can be read as characters, and file data created as characters can be read as records. The only difference between character and record I/O is that records must end with a $9B value. $9B is treated as ordinary data when using character I/O.

BASIC supports record I/O quite well. The commands PRINT and INPUT can be used to write and read records from files. BASIC does not completely support character I/O. The commands GET and PUT allow you to read and write a single byte at a time. However, the OS has the ability to read and write blocks of characters. This ability is not used by BASIC. In using this feature in the OS, you must specify the length and address of the data block to be transferred. To use the character block mode of the OS from BASIC, you can write an assembly language module to be called from BASIC by the USR function. Figure 8-16 has an example of a subroutine to do character block I/O.

The XIO command in BASIC is a general Input/Output statement that allows for direct communication with CIO from BASIC., It is described in more detail in the following subsection.
Disk Utility Package
The Disk Utility Package (DUP) is a set of utilities for diskette management, familiarly seen as the DOS menu. DUP executes commands by calling FMS through CIO. The commands are:

```
A. DIRECTORY
B. RUN CARTRIDGE
C. COPY FILES
D. DELETE FILES
E. RENAME FILES
F. LOCK FILES
G. UNLOCK FILES
H. WRITE DOS FILES
I. FORMAT DISK
J. DUPLICATE DISK
K. SAVE BINARY FILE
```

```
L. LOAD BINARY FILE
M. RUN AT ADDRESS
N. WRITE MEM.SAV FILE
O. DUPLICATE FILE
```

The following subsections describe each of these functions. However, for complete information on these functions, refer to the DISK OPERATING SYSTEM II Manual.

Wild Cards

Many of the DUP commands require a filename specification. DOS recognizes two "wild cards" that you can substitute for characters in a filename. Wild cards are represented by the special characters, question mark (?) and asterisk (*).

These characters are used in filename descriptors where, for whatever reason, there exists some uncertainty as to the exact filename. An example of this would be when a filename extension is not known, for instance. Another example would be when you want to copy only files with a specific extension such as OBJ.

The question mark (?) may be substituted for a single character. The asterisk (*) can stand for any valid combination of characters or number of characters. The following examples illustrate the use of these characters in a Directory command.

*.BAS will list all files on a diskette in Drive 1 that end in BAS.
D2:*.* will list all the program files on the Drive 2 diskette.
PRO*.BAS will list all the program files on diskette in Drive 1 that begin with PRO and have BAS as the extender.
TEST?? will list all the program files on diskette in Drive 1 that begin with TEST and have any combination of letters or numbers for the last two characters.

Disk Directory (A)

The Disk Directory contains a list of all the files on a diskette. This command will list the filenames, the extender, and the number of sectors that the file occupies on the diskette. A partial list can be generated by entering specific filename parameters. Wild cards can be used in the parameters.

RUN CARTRIDGE (B)

The 'B' command, Run Cartridge, is typed, DOS gives control of the system to whichever cartridge is inserted. The response from that point on is dependent upon specific cartridges. BASIC, for instance, will respond by printing READY on the screen.

If the diskette in drive 1 has not been changed since the DUP was loaded, and if a MEM.SAV file is present on the diskette, then the contents of this file is copied back into RAM before control is released to the cartridge. This file normally is used to store the contents of the portion of RAM that DUP occupies when it is loaded. However, this file must already exist on the diskette when a call is issued to load DUP. Before DUP is loaded, the RAM contents are written out to the diskette for later retrieval. You can think of MEM.SAV and DUP as swapping places between the diskette and RAM.

Copy File (C)

The Copy File command ('C'), is used to copy a file from a diskette in one disk drive to another diskette in a second disk drive. You will be prompted to give file specifications for the file to COPY-FROM, TO. The first file specification may contain wild cards, and can be used to indicate a series of files to be copied. The second parameter is also generally a file specification, but may also be a destination device such as E: (screen) or P: (printer). The second parameter may be followed with a '/A? option, which indicates that the first named file should be appended to the second file. This option should not be used with tokenized Basic files.

Delete File (D)

This option allows you to delete one or more files from a diskette. Wildcards can be used in file specification names. You can avoid having to respond to the delete verification promt by appending a '/N' option to the file specification.

Rename File (E)

This option allows you to change the name of an existing file on a diskette. You must provide two parameters, OLD NAME and NEW. The first parameter must be a complete file specification, but the second is just the new file name. Wild cards are permitted for both names. If no device specification is included, D1: is assumed. An error will be generated if the first file name doesn't exist on the diskette, if the file is locked, or if the diskette is write protected.

Lock File (F)

This command is used to prevent a file from being inadvertently erased or modified. A locked file is indicated by an asterisk (*) preceding its name in the directory. Note however, that the Format command pays no attention to the Lock status of any file on a diskette.

Unlock File (G)

This option is used to Unlock a file that has been previously Locked. Both this and the Lock commands may use wild cards.

Write DOS File (H)

This option must be used to create a copy of DOS on a formatted diskette, since they can't be copied with a Copy command.

Format Diskette (I)

This option is used to create the sector and track information on a blank diskette so that it may be used by DOS. If a bad sector is encountered during the formatting process, DOS will not continue. A diskette with a bad sector cannot be formatted, and is useless. WARNING! Be very careful with this command, for it will wipe out any existing file on the diskette.

Duplicate Disk (J)

This menu option is used to create an exact duplicate of any diskette that has been created and maintained by DOS. This option can be used with either single or multiple drive systems. Duplicating on a single drive system does require repeated manual swapping of the source and the destination diskettes.

The duplication process occurs on a sector-by-sector basis. However, only those sectors that are marked as in use by the Volume Table of Contents are copied.

Care should be taken in using this command, for it destroyes any files that may have resided on the destination diskette. A good policy would be to place a write protect tab on the source diskette to preclude a catastrophic mistake by typing the source and destination values in reverse order.

Binary Save (K)

This command is use to save the contents of memory locations in an object file (binary) format. This format is also used by the ATARI Editor Assembler Cartridge. This format consists of two header bytes of $FF, two bytes for the starting load address, and two bytes for the ending load address. The remainder of the file is actual load data. You will be prompted to enter a filename and the starting and ending addresses for the load. There is also two additional address values that may optionally be entered. These are values that upon load will be placed In locations known as INIT [$02EO,2] and RUN [$02E2,2]. If these locations are updated during a load, then the code pointed to by the values in these locations will be executed.

Binary Load (L)

This command is used to load a binary load file from the diskette into RAM memory. If values for INIT and RUN values have been appended to this file, then it will function as a load-and-go routine.

Run At Address (M)

This command is used to transfer control to a machine language routine located in memory. This is normally used to start a program that has been loaded, but did not have INIT or RUN values appended to the file.

Create MEM.SAV (N)

This menu option is used to create a file called MEM.SAV. This file is used to save the contents of memory that will be overlaid when the DUP is loaded in. Effectively then, MEM.SAV and DUP swap places from RAM to disk. Note that MEM.SAV must be on the diskette in drive 1 to work. It also takes about 20 seconds to swap memory out and load DUP in if MEM.SAV is used.

Duplicate File (O)

This option is provided to copy a file from one diskette to another in a single drive system. Functionally, it is very similar to the single drive Copy command. The primary difference is that Duplicate file can be used to copy a file created under DOS 1, whereas the Copy command cannot.

Substituting the XIO Command for DUP Menu Options

The XIO command in BASIC is a general I/O statement that issues a direct call to the Centralized Input/Output Subsystem. The format of the XIO command is:

```
XIO command number, #iocb, auxialiary 1, auxialiary 2, file specification
```

The XIO command can be used to perform functions that would normally require DUP to be present. The command number for various DUP functions are shown below.

COMMAND NUMBER	FUNCTION
3	OPEN
5	GET Record
7	GET Characters
9	PUT Record
11	PUT Characters
12	CLOSE
13	STATUS Request
32	RENAME
33	DELETE
35	LOCK File
36	UNLOCK File

Random Access
One common use of the diskette is to store records that may be accessed n an arbitrary order. This is referred to as Random Access. Using the I/O commands in conjunction with the special commands NOTE and POINT allows you to create and use random access files.

DOS keeps a file pointer for each file currently OPEN which tells the DOS the location of the next byte to be accessed in the file. NOTE and POINT are used to find out the current value of this pointer, or to set it to a specific value. The file pointer has two parameters, a sector number and a byte number. The sector number is a value from 1-719 that tells DOS what sector on the diskette the file pointer is pointing to. The byte number indicates the specific byte in the sector that will be the next accessed. Figure 9-2 shows the value of the file pointer at different bytes within the file. All values are hex. The file pointer values for the bytes in this file are given below the bytes in the file.

File	A 41	B 42	C 43	EOL 9B	D 44	E 45	F 46	EOL 9B	G 47	H 48	I 49	J 4A	K 4B	EOL 9B	...	A 41	B 42
File pointer																	
Sector Number	50	50	50	50	50	50	50	50	50	50	50	50	50	50	...	50	51
Byte Count	0	1	2	3	4	5	6	7	8	9	A	B	C	D	...	7C	0

Figure 9-2 NOTE and POINT Values

The above file was created in BASIC by the following program:

```
10 OPEN #1,8,0,"D:FILE"
20 ?#l;"ABC"
30 ?#1;"DEF"
40 ?#1;"GHIJK"
    :
    :         :REM Fill the rest of the sector
    :
100 ?#1;"AB"  :REM This writes a record that crosses end of sector
150 CLOSE #1
```

The sector number of 50 was arbitrarily chosen for this example. The sector number changed to 51 because the file is longer than a single sector. FMS linked the file to the next available sector, 51. The record "AB" crosses the end of the first sector.

The byte count of the file pointer starts at 0 and is incremented until the end of the sector, $7D (125 Decimal). DOS reserves the last 3 bytes of every sector for overhead data for the file. The maximum byte number is 124 (0-124 = 125 total bytes). When the file reaches the end of a sector, the byte number recycles to 0.

When the POINT command is used to set the file pointer, DOS checks that the sector pointed to belongs to the file that is OPENed. If a file number mismatch is found, the POINT operation is not allowed.

Figure 9-3 is a subroutine that may be used to save records, keep track of where they are, and retrieve them in random access fashion.

```
1000 REM THIS ROUTINE CREATES AND ACCESSES RANDOM ACCESS FILES FOR FIXED
1001 REM LENGTH RECORDS
1002 REM
1003 REM ... COMMANDS ARE
1004 REM CMD=1 WRITE  NTH RECORD
1005 REM CMD=2 READ   NTH RECORD
1006 REM CMD=3 UPDATE NTH RECORD
1007 REM
1008 REM RECORD$ IS THE INPUT/OUTPUT RECORD
1009 REM N IS THE RECORD NUMBER
1010 REM INDEX IS A TWO DIMENSIONAL ARRAY DIM'ED INDEX(1,RECNUM)
1015 REM INDEX HOLDS THE NOTE VALUES FOR ALL RECORDS
1020 REM THIS ROUTINE ASSUMES LOGICAL FILE #1 HAS BEEN OPENED FOR I/O
1100 REM
1120 REM ROUTINE BEGINS AT 1200
1130 REM
1200 ON CMD GOTO 2000,3000,4000
2000 REM ............................
2100 REM WRITE NTH RECORD
2200 NOTE #1,X,Y
2300 INDEX(SEC,N)=X:INDEX(BYTE,N)=Y
2400 ? #1;RECORD$:RETURN
3000 REM ............................
3010 REM READ NTH RECORD
3020 REM
```

```
3030 X=INDEX(SEC,N):Y=INDEX(BYTE,N)
3040 POINT #1,X,Y
3050 INPUT #1;RECORD$
3060 RETURN
4000 REM ............................
4010 REM UPDATE NTH RECORD
4020 REM
4040 X=INDEX(SEC,N):Y=INDEX(BYTE,N)
4050 POINT #1,X,Y
4060 ? #1;RECORD$
4070 RETURN
```

Figure 9-3 NOTE and POINT Example

FMS Disk Utilization

The map below shows the overall layout that DOS uses in managing disk sector utilization for a standard 720 sector diskette.

```
+--------------+
| BOOT record  |  Sector 1
+--------------+
|  FMS BOOT    |  Sector 2
=    file      =
|  'DOS.SYS'   |  Sector 40 ($28)
+--------------+
|     Disk     |  Sector 41 ($29)
=   Utilities  =
|    Package   |  Sector 83 ($53)
+--------------+
|     User     |  Sector 84 ($54)
=     file     =
|     Area     |  Sector 359 ($167)
+--------------+
| VOLUME TABLE |  Sector 360 ($168)
| of CONTENTS  |
+--------------+
|     File     |  Sector 361 ($169)
=  Directory   =
|              |  Sector 368 ($170)
+--------------+
|     User     |  Sector 369 ($171)
=     file     =
|     Area     |  Sector 719 ($2CF)
+--------------+
|    Unused    |  Sector 720 ($2D0)
+--------------+
```

FMS Boot Record

The first sector on a diskette is reserved for FMS boot usage. This record contains information concerning the FMS system configuration, as well as an indication of whether the DOS.SYS is present on the diskette or not.

If the DOS files are present they usually begin at sector 2 and extend for 81 sectors.

Volume Table of Contents

Sector 360 is reserved for the FMS Volume Table of Contents (VTOC). This table contains a bit map that shows which sectors on the diskette are allocated and which are free. Since VTOC is referred to before every disk write, sector 360 was chosen to hold VTOC. This sector is in the middle of the diskette and has the minimum average access time of any diskette sector. The bit map begins in byte 10 of VTOC and extends to byte 99. Each byte in this area contains allocation information for eight sectors. A 0 in a bit position indicates the corresponding sector is in use, and a 1 Indicates it is available. The volume bit map is organized as shown below:

```
 7             0
+-+-+-+-+-+-+-+-+
|  1 2 3 4 5 6 7|  Byte 10 of VTOC
+-+-+-+-+-+-+-+-+
|8 9 . . . . . .|  11
=               =
|               |  99
```

```
+-+-+-+-+-+-+-+-+
```

File Directory Format

Eight sectors (361-368) are reserved for a diskette file directory, with each sector able to store information for up to eight files. Thus, the maximum number of files that can be placed on a single diskette is 64.

Each file directory entry consists of 16 bytes. The format of each entry is shown on the next page.

```
+---------------+
|   flag byte   | Byte 0
+---------------+
|  sector  (lo) |      1
+               +
|    count (hi) |      2
+---------------+
| starting (lo) |      3
+  sector       +
|  number  (hi) |      4
+---------------+
|           (1) |      5
+               +
|           (2) |      6
+               +
|           (3) |      7
+               +
|  file     (4) |      8
+               +
|  name     (5) |      9
+               +
|  primary  (6) |     10
+               +
|           (7) |     11
+               +
|           (8) |     12
+---------------+
|  file     (1) |     13
+               +
|  name     (2) |     14
+               +
| extension (3) |     15
+---------------+
```

The flag byte has the following bit assignments:
Bit-7 = 1 if the file has been deleted.
Bit-6 = 1 if the file is in use.
Bit-5 = 1 if the file is locked.
Bit-0 = 1 if OPEN for output.

FMS File Sector Format

The format of a sector in a diskette file is shown below:

```
 7             0
+-+-+-+-+-+-+-+-+
|               | Byte 0
=   D A T A     =
|               |    124
+-+-+-+-+-+-+-+-+
|  file     |hi |    125
+-+-+-+-+-+-+-+-+
|forward pointer|    126
+-+-+-+-+-+-+-+-+
|S| byte count  |    127
+-+-+-+-+-+-+-+-+
```

The file # is information that FMS uses to ensure file integrity is maintained. This field contains the value of the directory position for that file. If there is ever a mismatch between the file's position in the directory and the file number field in a sector, the FMS will generate an error, and abort whichever operation was being performed.

The forward pointer is a 10-bit pointer that indicates the next sector in a file. This is described as a forward linked list, with the forward pointer of the last sector equal to 0.

If the S bit is set (i.e., equal to 1), then the sector is a "short sector" and contains less than 125 data bytes.

The byte count field contains the number of bytes in the sector.

THE AUTORUN.SYS FILE
DOS contains a feature that allows a special file to be loaded into memory each time the system is powered-up. This can be data to customize features of the system such as setting up different margin values, or changing the default colors. If desired this can be a machine language program to be executed before the normal DOS boot process occurs.

This file must be 6 binary load file with the name AUTORUN.SYS. To make this an executable file, an address value must be loaded into the INIT [$02E0,2] or the RUN [$02E2,2] locations. The difference between these two parameters is that the code pointed to by INIT will be executed as soon as that location is loaded, whereas code pointed to by RUN will only be executed after the load process is complete. To return control to DOS after executing an AUTORUN.SYS file, terminate your code with an RTS.

The AUTORUN.SYS file can be extremely useful in setting up "load- and-go" assembly language routines. It also provides a method of reconfiguring the OS by "stealing" certain of the system vectors before DOS has a chance to be initialized. Among other things, this feature can be used to provide a certain measure of diskette protection. Refer to Figure 8-3 for an example of setting up an AUTORUN.SYS file to reset the MEMLO pointer.

Chapter X

Atari Basic

10 Atari Basic

WHAT IS ATARI BASIC?

ATARI BASIC is an interpreted language. This means programs can be run when they are entered without intermediate stages of compilation and linking. The ATARI BASIC interpreter resides in an 8K ROM cartridge in the left slot of the computer. It encompasses addresses A000 through BFFF. At least 8K of RAM is required to run BASIC.

To use ATARI BASIC effectively, you must know its strengths and weaknesses. With this information, programs can be written that make good use of the assets and features of ATARI BASIC.

Strengths of ATARI BASIC

- It supports the operating system graphics - Simple graphics calls can be made to display information on the screen.
- It supports the hardware - Such calls as SOUND, STICK and PADDLE are simple interfaces to the hardware of the computer.
- Simple assembly interface - The USR function allows easy user access to assembly language routines.
- ROM based interpreter - The BASIC interpreter is in ROM, which prevents accidental modification by the user program.
- DOS support - Specialized calls such as NOTE and POINT (DOS 2.0S) allow the user to randomly access a disk through the disk operating system.
- Peripheral support - Any peripheral recognized by the operating system can be accessed from a BASIC program.

Weaknesses of ATARI BASIC

- No support of integers - All numbers are stored as 6- byte BCD floating point numbers.
- Slow math package - Since all numbers are six bytes long, math operations become rather slow.
- No string arrays - Only one-dimensional strings can be created.

HOW ATARI BASIC WORKS

The workings of the BASIC interpreter are summarized as follows:

- BASIC gets a line of input from the user and converts it into a tokenized form.
- It then puts this line into a token program.
- This program is then executed.

The details of these operations are discussed in the following four sections.

- The Tokenizing Process
- The Token File Structure
- The Program Execution Process
- System Interaction

THE TOKENIZING PROCESS

In simple terms, the tokenization of a line of code in BASIC looks like this:

1. BASIC gets a line of input
2. It then checks for legal syntax
3. During syntax checking it is tokenized
4. The tokenized line is moved into the token program
5. If the line is in immediate mode it is executed

To better understand the tokenizing process, some terms must first be defined:

Token	An 8-bit byte containing a particular interpretable code.

Statement	The first executable token of a statement that tells BASIC to interpret the tokens that follow in a particular way.
Line	A complete "sentence" of tokens that causes BASIC to perform some meaningful task. In LIST form, statements are separated by colons.
Command	One or more statements preceded either by a line number in the range of 0 to 32767 or an immediate mode line with no number.
Variable	A token that is an indirect pointer to its actual value; can be changed without changing the token.
Constant	A 6-byte BCD value preceeded by a special token. This value remains unchanged throughout program execution.
Operator	Any one of 46 tokens that in some way move or modify the values that follow them.
Function	A token that when executed returns a value to the program.
EOL	End of Line. A character with the value 9B hex.
BCD	Binary code decimal. A number that uses the 6502 decimal mode.

BASIC begins the tokenizing process by getting a line of input. This input will be obtained from one of the handlers of the operating system. Normally it is from the screen editor; however with the ENTER command, any device can be specified. The call BASIC issues is a GET RECORD command, and the data returned is ATASCII information terminated by an EOL. This data is stored by CIO into the BASIC Input Line Buffer from 580 to 5FF hex.

After the record is returned, the syntax checking and tokenizing processes begin. First BASIC looks for a line number. If one is found, it is converted into a 2- byte integer. If no line number is present, it is assumed to be in immediate mode and the line-number 8000 hex is assigned to it. These will be the first two tokens of the tokenized line. This line is built in the token output buffer that is 256 bytes long and resides at the end of the reserved operating system RAM.

The next token is a dummy byte reserved for the byte count (or offset) from the start of this line to the start of the next line. Following that is another dummy byte for the count of the start of this line to the start of the next statement. These values will be set when tokenization is complete for the line and the statement respectively. The use of these values is discussed in the program execution process section.

BASIC now looks for the command of the first statement of the input line. A check is made to determine if this is a valid command by scanning a list of legal commands in ROM. If a match is found, then the next byte in the token line becomes the number of the entry in the ROM list that matched. If no match is found, a syntax error token is assigned to that byte and BASIC stops tokenizing, copies the rest of the input buffer in ATASCII format to the token output buffer, and prints the error line.

Assuming a good line, one of seven items can follow the command: a variable, a constant, an operator, a function, a double quote, another statement, or an EOL. BASIC tests if the next input character is numeric. If not then it compares that character and those following against the entries of the variable name table. If this is the first line of code entered in the program then no match is found. The characters are then compared against the function and operator tables. If no match is found there then BASIC assumes that this is a new variable name. Since this is the first variable it is assigned the first entry in the variable name table. The characters are copied out of the input buffer and stored into the name table with the most significant bit (MSB) set on the last byte of the name. Eight bytes are then reserved in the variable value table for this entry. (See the variable value table discussion in the section, "Token File Structure".)

The token that ends up in the tokenized line is the variable number minus one; with the MSB set. Thus the token of the first variable entered would be 80 Hex, the second would be 81, and so on up to FF for a total of 128 unique variable numbers.

If a function is found, then its entry number in the operator function table is assigned to the token. Functions require certain sequences of parameters; these are contained in syntax tables, and if they are not matched, a syntax error will result.

If an operator is found, then a token is given its table entry number. Operators can follow each other in a rather complex fashion (such as multiple parentheses), so the syntax checking of them is a bit complicated.

In the case of the double quotes, BASIC assumes that a character string is following and assigns a 0F hex to the output token and reserves a dummy byte for the string length. The characters are moved from the input buffer into the output buffer until the second set of quotes is found. The length byte is then set to the character count.

If the next characters in the input buffer are numeric, BASIC converts them into a 6-byte BCD constant. A 0E hex token will be put in the output buffer, followed by the six byte constant.

When a colon is encountered, a 14 hex token is inserted in the output buffer and the offset from the start of the line is stored in the dummy byte that was reserved for the count to the start of the next statement. At this point another dummy byte is reserved and the process goes back to get a command.

When the EOL is found, a 16 hex token is stored and the offset from the start of the line is put in the dummy byte for the line offset. At this point, tokenization is complete and BASIC moves the token line into the token program. First it searched the program for that line number. If it is found it replaces the old line with the new one. If it is not found, then it inserts the new line in the correct numerical sequence. In both cases, the data following the line will be moved either up or down in memory to allow for an expanding and contracting program size.

BASIC now checks if the tokenized line is an immediate mode line. If so, that line is executed according to the methods described in the interpretive process; if not, BASIC goes back to get another line of input.

If at any time during the tokenizing process the length of the token line exceeds 256 bytes, an ERROR 14 message (line too long) is sent to the screen and BASIC goes back to get the next line of input.

An example line of input and its token form looks like this (all token values are hexadecimal):

```
10 LET X=1 : PRINT X
```

0A 00	13	0F	06	80	2D	0E	40 01 00 00 00 00	14	13	20	80	16

Line 10	Line Offset	Statement Offset	Let	X	=	Numeric Constant	1	End Of Statement	Statement Offset	Print	X	End Of Line

Figure 10-1 Example Line of Tokenized Input

COMMANDS			OPERATORS			FUNCTIONS		
HEX	DEC		HEX	DEC		HEX	DEC	
00	00	REM	0E	14	[NUM CONST]	3D	61	STR$
01	01	DATA	0F	15	[STR CONST]	3E	62	CHR$
02	02	INPUT	10	16	[NOT USED]	3F	63	USR
03	03	COLOR	11	17	[NOT USED]	40	64	ASC
04	04	LIST	12	18	,	41	65	VAL
05	05	ENTER	13	19	$	42	66	LEN
06	06	LET	14	20	: [STMT END]	43	67	ADR
07	07	IF	15	21	;	44	68	ATN
08	08	FOR	16	22	[LINE END]	45	69	COS
09	09	NEXT	17	23	GOTO	46	70	PEEK
0A	10	GOTO	18	24	GOSUB	47	71	SIN
0B	11	GO TO	19	25	TO	48	72	RND
0C	12	GOSUB	1A	26	STEP	49	73	FRE
0D	13	TRAP	1B	27	THEN	4A	74	EXP
0E	14	BYE	1C	28	#	4B	75	LOG
0F	15	CONT	1D	29	<= [NUMERICS]	4C	76	CLOG
10	16	COM	1E	30	<>	4D	77	SQR
11	17	CLOSE	1F	31	>=	4E	78	SGN
12	18	CLR	20	32	<	4F	79	ABS
13	19	DEG	21	33	>	50	80	INT
14	20	DIM	22	34	=	51	81	PADDLE
15	21	END	23	35	*	52	82	STICK
16	22	NEW	24	36	+	53	83	PTRIG
17	23	OPEN	25	37	-	54	84	STRIG
18	24	LOAD	26	38	/			
19	25	SAVE	27	39	NOT			
1A	26	STATUS	28	40	OR			
1B	27	NOTE	29	41	AND			
1C	28	POINT	2A	42	(			
1D	29	XIO	2B	43	)			
1E	30	ON	2C	44	= [ARITHM ASSIGN]			
1F	31	POKE	2D	45	= [STRING ASSIGN]			
20	32	PRINT	2E	46	<= [STRINGS]			
21	33	RAD	2F	47	<>			
22	34	READ	30	48	>=			
23	35	RESTORE	31	49	<			
24	36	RETURN	32	50	>			
25	37	RUN	33	51	=			
26	38	STOP	34	52	+ [UNARY]			
27	39	POP	35	53	-			
28	40	?	36	54	([STRING LEFT PAREN]			
29	41	GET	37	55	([ARRAY LEFT PAREN]			
2A	42	PUT	38	56	([DIM ARRAY LEFT PAREN]			
2B	43	GRAPHICS	39	57	([FUN LEFT PAREN]			
2C	44	PLOT	3A	58	([DIM STR LEFT PAREN]			
2D	45	POSITION	3B	59	, [ARRAY COMMA]			
2E	46	DOS	3C	60				
2F	47	DRAWTO						
30	48	SETCOLOR						
31	49	LOCATE						
32	50	SOUND						
33	51	LPRINT						
34	52	CSAVE						
35	53	CLOAD						
36	54	[IMPLIED LET]						
37	55	ERROR - [SYNTAX]						

THE TOKEN FILE STRUCTURE
The token file contains two major segments: (1) a group of zero page pointers that point into the token file, and (2) the actual token file itself. The zero page pointers are 2-byte values that point to various sections of the token file. There are nine 2-byte pointers and they are in locations 80 to 91 hex. Following is a list of the pointers and the sections of the token file they reference.

Pointer (hex)	Token File Section (Contiguous Blocks)
LOMEM 80,81	Token output buffer - This is the buffer BASIC uses to tokenize one line of code. It is 256 bytes long. This buffer resides at the end of the operating system's allocated RAM.
VNTP 82,83	Variable name table - A list of all the variable names that have been entered in the program. They are stored as ATASCII characters, each new name stored in the order it was entered. Three types of name entries exist: **1. Scalar variables** - MSB set on last character in name. **2. String variables** - last character is a with the MSB set. **3. Array variables** - last character is a with the MSB set.
VNTD 84,85	Variable name table dummy end - BASIC uses this pointer to indicate the end of the name table. This normally points to a dummy zero byte when there are less than 128 variables. When 128 variables are present, this points to the last byte of the last variablename.

<table>
<tr><td>VVTP
86,87</td><td>Variable value table - This table contains current information on each variable. For each variable in the name table, eight bytes are reserved in the value table. The information for each variable type is:
<table>
<tr><td>Byte Number</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Scalar</td><td>00</td><td>Var#</td><td colspan="6">6-byte BCD constant</td></tr>
<tr><td>Array (DIMed)
(unDIMed)</td><td>41
40</td><td>Var#</td><td colspan="2">Offset from
STARP(8C,8D)</td><td colspan="2">first
DIM + 1</td><td colspan="2">second
DIM + 1</td></tr>
<tr><td>String (DIMed)
(unDIMed)</td><td>81
80</td><td>Var#</td><td colspan="2">Offset from
STARP</td><td colspan="2">Length</td><td colspan="2">DIM</td></tr>
</table>
A scalar variable contains a numeric value. An example is X=1. The scalar is X and its value is 1, stored in 6-byte BCD format. An array is composed of numeric elements stored in the string/array area and has one entry in the value table. A string, composed of character elements in the string/array area, also has one entry in the table. The first byte of each value entry indicates the type of variable: 00 for a scalar, 40 for an array, and 80 for a string. If the array or string has been dimensioned, then the LSB is set on the first byte. The second byte contains the variable number. The first variable entry is number zero, and if 128 variables were present, the last would be 7F.

In the case of the scalar variable the third through eighth byte contain the 6-byte BCD number that has currently been assigned to it. For arrays and strings, the third and fourth bytes contain an offset from the start of the string/array area (described below) to the beginning of the data.

The fifth and sixth bytes of an array contain its first dimension. The quantity is a 16-bit integer and its value is 1 greater than the user entered. The seventh and eighth bytes are the second dimension, also a value of 1 greater.

The fifth and sixth bytes of a string are a 16 bit integer that contains its current length. The seventh and eighth bytes are its dimension (up to 32767 bytes in size).</td></tr>
<tr><td>STMTAB
88,89</td><td>Statement Table - This block of data includes all the lines of code that have been entered by the user and tokenized by BASIC, and it also includes the immediate mode line. The format of these lines is described in the tokenized line example of the section on the tokenizing process.</td></tr>
<tr><td>STMCUR
8A,8B</td><td>Current Statement - This pointer is used by BASIC to reference particular tokens within a line of the statement table. When BASIC is waiting for input, this pointer is set to the beginning of the immediate mode line.</td></tr>
<tr><td>STARP
8C,8D</td><td>String/Array area - This block contains all the string and array data. String characters are stored as one byte ATASCII entries, so a string of 20 characters will require 20 bytes. Arrays are stored with 6-byte BCD numbers for each element. A 10-element array would require 60 bytes. This area is allocated and subsequently enlarged by each dimension statement encountered, the amount being equal to the size of a string dimension or six times the size of an array dimension.</td></tr>
</table>

RUNSTK 8E,8F	Run time stack - This software stack contains GOSUB and FOR/NEXT entries. The GOSUB entry consists of four bytes. The first is a 0 byte indicating GOSUB, followed by the 2-byte integer line number on which the call occurred. This is followed by the offset into that line so the RETURN can come back and execute the next statement. The FOR/NEXT entry contains 16 bytes. The first is the limit the counter variable can reach. The second byte is the step or counter increment. Each of these quantities is in 6-byte BCD format. The thirteenth byte is the counter variable number with the MSB set. The fourteenth and fifteenth bytes are the line number, and the sixteenth is the line offset to the FOR statement.
MEMTOP 90,91	Top of application RAM - This is the end of the user program. Program expansion can occur from this point to the end of free RAM, which is defined by the start of the display list. The FRE function returns the amount of free RAM by subtracting MEMTOP from HIMEM (2E5,2E6). Note that the BASIC MEMTOP is not the same as the OS variable called MEMTOP.

THE PROGRAM EXECUTION PROCESS
Executing a line of code is a process that involves reading the tokens that were created during the tokenization process. Each token has a particular meaning that causes BASIC to execute a specific series of operations. The method of doing this requires that BASIC get one token at a time from the token program and then process it. The token is an index into a jump table of routines, so a PRINT token will point indirectly to a PRINT processing routine. When that processing is complete, BASIC returns to get the next token. The pointer that is used to fetch each token is called STMCUR and is at 8A and 8B.

The first line of code that is executed in a program is the immediate mode line. This is usually a RUN or GOTO. In the case of the RUN, BASIC gets the first line of tokens from the statement table (tokenized program) and processes it. If all the code is in-line, then BASIC merely executes consecutive lines.

If a GOTO is encountered, then the line to go to must be found. The statement table contains a linked list of tokenized BASIC lines. These lines are stored in ascending numerical order. To find a line somewhere in the middle of the table, BASIC starts by finding the first line of the program.

The address of the first line is contained in the STMTAB pointer at 88 and 89. This address is now stored in a temporary pointer. The first 2 bytes of the first line are its line number which is compared against the requested line number. If the first number is less, then BASIC gets the next line by adding the third byte of the first line to the temporary pointer. The temporary pointer will now be pointing to the second line. Again the first 2 bytes of this new line are compared to he requested line, and if they are less, the third byte is added to the pointer. If a line number does match, the contents of the temporary pointer are moved into STMCUR and BASIC fetches the next token from the new line. Should the requested line number not be found, an ERROR 12 is generated.

The GOSUB involves more processing than the GOTO. The line finding routine is the same, but before BASIC goes to that line it sets up an entry in the Run Time Stack. It allocates four bytes at the end of the stack and stores a 0 in the first byte to indicate a GOSUB stack entry. It then stores the line number it was on when the call was made into the next two bytes of the stack. The final byte contains the offset in bytes from the start of that line to where the GOSUB token was found. BASIC then executes the line it looked up. When the RETURN is found, the entry on the stack is pulled off, and BASIC returns to the calling line.

The FOR command causes BASIC to allocate 16 bytes on the Run Time Stack. The first six bytes are the limit the variable can reach in 6-byte BCD format. The second six bytes are the step, in the same format. Following these, BASIC stores the variable number (MSB set) of the counting variable. It then stores the present line number (two bytes) and the offset into the line. The rest of the line is then executed.

When BASIC finds the NEXT command, it looks at the last entry on the stack. It makes sure the variable referenced by the NEXT is the same as the one on the stack and checks if the counter has reached or exceeded the limit. If not then BASIC returns to the line with the FOR statement and continues execution. If the limit was reached, then the FOR entry is pulled off the stack and execution continues from that point.

When an expression is evaluated, the operators are put onto an operator stack and are pulled off one at a time and evaluated. The order in which the operators are put onto the stack can either be implied, in which case BASIC looks up the operator's precedence from a ROM table, or the order can be explicitly stated by the placement of parentheses.

Pressing the BREAK key at any time causes the operating system to set a flag to indicate this occurrence. BASIC checks this flag after each token is processed. If it finds it has been set, it stores the line number at which this occurred, prints out a "STOPPED AT LINE XXXX" message, clears the BREAK flag and waits for user input. At this point the user could type CONT and program execution would continue at the next line.

SYSTEM INTERACTION
BASIC communicates with the Operating System primarily through the use of 1/0 calls to the Central 1/0 Utility (CIO). Following is a list of user BASIC calls and the corresponding operating system IOCB (Input/Output Control Block) setups.

BASIC	OS
`OPEN #1,12,0,"E:"`	IOCB=1 Command=3 (OPEN) Aux1=12 (Input/Output) Aux2=0 Buffer Address=ADR("E:")
`GET #1,X`	IOCB=1 Command=7 (Get Characters) Buffer Length=0 Character returned in accumulator
`PUT #1,X`	IOCB=1 Command=11 (Put Characters) Buffer Length=0 Character output through accumulator
`INPUT #1,A$`	IOCB=1 Command=5 (Get Record) Buffer Length=Length of A$ (not over 120) Buffer Address=Input Line Buffer
`PRINT #1,A$`	IOCB=1 BASIC uses a special put byte vector in the IOCB to talk directly to the handler.
`XIO 18,#6,12,0,"S:"`	IOCB=6 Command=18 (Special - Fill) Aux1=12 Aux2=0

SAVE/LOAD: When a BASIC token program is saved to a device, two blocks of information are written. The first block consists of seven of the nine zero page pointers that BASIC uses to maintain the token file. These are LOMEM(80,81) through STARP (8C,8D). There is one change made to these pointers when they are written out: The value of LOMEM is subtracted from each of the 2-byte pointers, and these new values are written to the device. Thus the first 2-bytes written will be 0,0.

The second block of information written consists of the following token file sections: (1) The variable name table, (2) the variable value table, (3) the token program, and (4) the immediate mode line.

When this program is loaded into memory, BASIC looks at the OS variable MEMLO (2E7,2E8) and adds its value to each of the 2-byte zero page pointers as they are read from the device. These pointers are placed back on page zero and then the values of RUNSTK(8E,8F) and MEMTOP (90,91) are set to the value in STARP.

Next, 256 bytes are reserved in memory above the value of MEMLO to allocate space for the token output buffer. Then the token file information, consisting of the variable name table through the immediate mode line, is read in. This data is placed in memory immediately following the token output buffer.

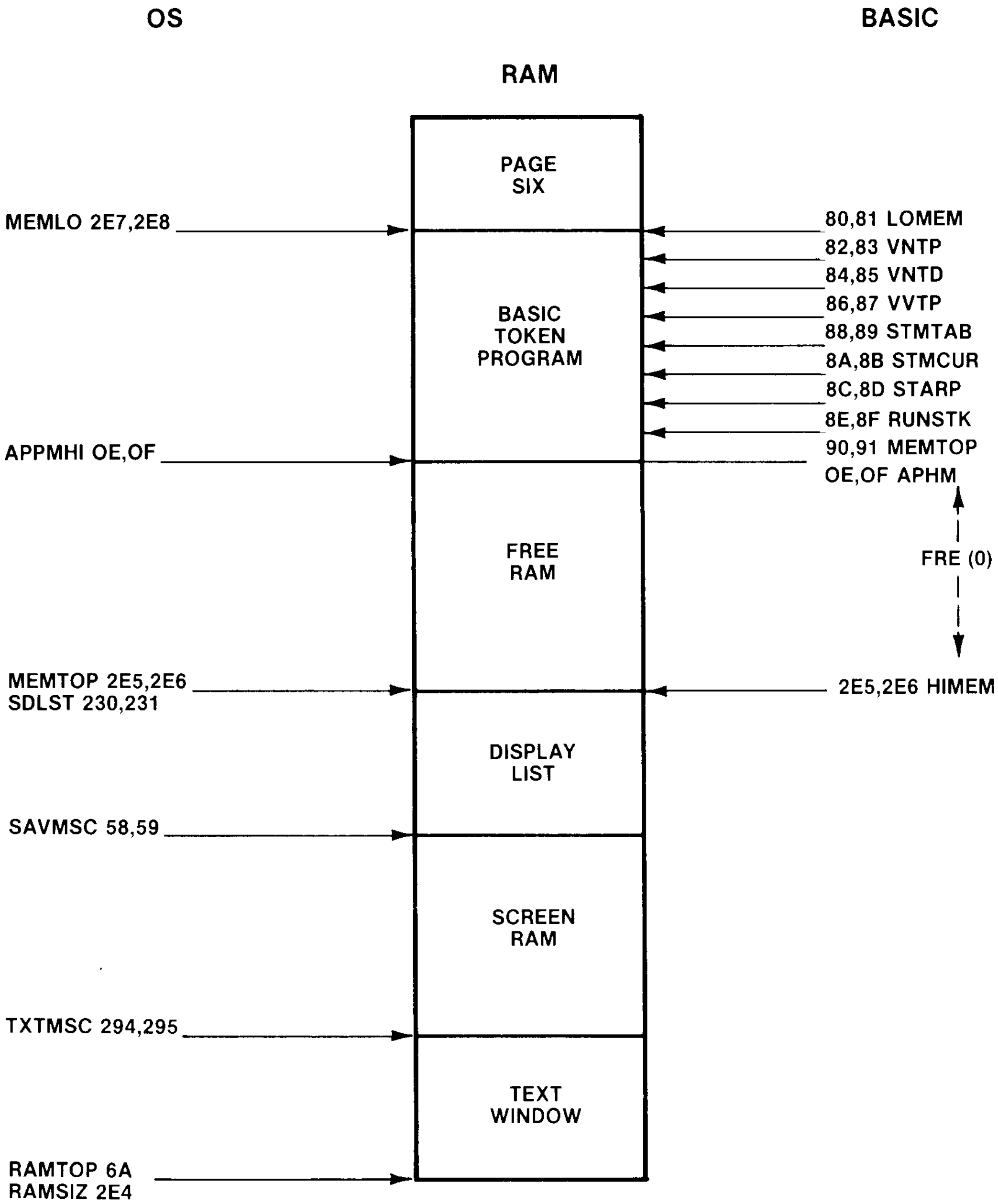


Figure 10-2 OS and BASIC Pointers (No DOS Present)

IMPROVING PROGRAM PERFORMANCE
Program performance can be improved in two ways. First the execution time can be decreased (it will run faster) and second, the amount of space required can be decreased, allowing it to use less RNA. To attain these two goals, the following lists can be used as guidelines. The methods of improvement in each list are primarily arranged in order of decreasing effectiveness. Therefore the method at the top of a list will have more impact than one on the bottom.

Speeding Up a BASIC Program

- Recode - Because BASIC is not a structured language, the code written in it tends to be inefficient. After many revisions it becomes even worse. Thus, the time spend to restructure the code is worthwhile.
- Check algorithm logic - Make sure that the code to execute a process is as efficient as possible.

- Put frequently called subroutines and FOR/NEXT loops at the start of the program - BASIC starts at the beginning of a program to look for a line number, so any line references near the end will take longer to reach.
- For frequently called operations within a loop use in-line code rather than subroutines - The program speed can be improved here since BASIC spends time adding and removing entries from the run time stack.
- Make the most frequently changing loop of a nested set the deepest - In this way, the run time stack will be altered the fewest number of times.
- Simplify floating point calculations within the loop - if a result is obtained by multiplying a constant by a counter, time could be saved by changing the operation to an add of a constant.
- Set up loops as multiple statements on one line - In this way the BASIC interpreter will not have to get the next line to continue the loop.
- Disable the screen display - If visual information is not important for a period of time, up to a 30 percent time savings can be made with a POKE 559,0.
- Use a faster graphics mode or a short display list - If a full screen display is not necessary then up to 25 percent time savings can be made.
- Use assembly code - Time savings can be made by encoding loops in assembler and using the USR function.

Saving Space In A BASIC Program

- Recode - As mentioned previously, restructuring the program will make it more efficient. It will also save space.
- Remove remarks - Remarks are stored as ATASCII data and merely take up space in the running program.
- Replace a constant used three times or more with a variable BASIC allocates seven bytes for a constant but only one for a variable reference, so six bytes can be saved each time a constant is replaced with a variable assigned to that constant's value.
- Initialize variables with a read statement - A data statement is stored in ATASCII code, one byte per character, whereas an assignment statement requires seven bytes for one constant.
- Try to convert numbers used once and twice to operations of predefined variables - An example is to define Z1 to equal 1, Z2 to equal 2, and if the number 3 is required, replace it with the expression Z1 + Z2.
- Set frequently used line numbers (in GOSUB and GOTO) to predefined variables - If the line 100 is referenced 50 times, approximately 300 bytes can be saved by equating Z100 to 100 and referencing Z100
- Keep the number of variables to a minimum - Each new variable entry requires 8 more bytes in the variable value table plus a few bytes for its name.
- Clean up the value and name tables - Variable entries are not deleted from the value and name tables even after all references to them are removed from the program. To delete the entries LIST the program to disk or cassette, type NEW, then ENTER the program.
- Keep variable names as short as possible - Each variable name is stored in the name table as ATASCII information. The shorter the names, the shorter the table.
- Replace text used repeatedly with strings - On screens with a lot of text, space can be saved by assigning a string to a commonly used set of characters.
- Initialize strings with assignment statements - An assignment of a string with data in quotes requires less space than a READ statement and a CHR$ function.
- Concatenate lines into multiple statements - Three bytes can be saved each time two lines are converted into two statements on one line.
- Replace once used subroutines with in-line code - The GOSUB and RETURN statements waste bytes if used only once.
- Replace numeric arrays with strings if the data values do not exceed 255 - Numeric array entries require six bytes each, whereas string elements only need one.
- Replace SETCOLOR statements with POKE commands - This will save 8 bytes.
- Use cursor control characters rather than POSITION statements The POSITION statement requires 15 bytes for the X,Y parameters whereas the cursor editing characters are one byte each.
- Delete lines of code via program control - See the advanced programming techniques

section.

- Modify the string/array pointer to load predefined data - By changing the value in STARP, string and array information can be saved.
- Small assembly routines can be stored in USR calls - For example X=USR(ADR("hhh*LVd"),16).
- Chain programs - An example would be an initialization routine that is run first and then loads and executes the main program.

ADVANCED PROGRAMMING TECHNIQUES
An understanding of fundamentals of ATARI BASIC makes it possible to write some interesting applications. These can be strictly BASIC operations, or they can also involve features of the operating system.

Example 1 - String Initialization - This program will set all the bytes of a string of any length to the same value. BASIC copies the first byte of the source string into the first byte of the destination string, then the second, third, and so on. By making the destination string the second byte of the source, the same character can be stored throughout the entire string.

```
10 REM STRING INITIALIZATION
20 DIM A$(1000)
30 A$(l)="A":A$(1000)="A"
40 A$(2)=A$
```

Example 2 - Delete Lines Of Code - By using a feature of the operating system, a program can delete or modify lines of code within itself. The screen editor can be set to accept data from the screen without user input. Thus by first setting up the screen, positioning the cursor to the top, and then stopping the program, BASIC will be getting the commands that have been printed on the screen.

```
10 REM DELETE LINE EXAMPLE
20 GRAPHICS 0:POSITION 2,4
30 ? 70:? 80:? 90:? "CONT"
40 POSITION 2,0
50 POKE 842,13:STOP
60 POKE 842,12
70 REM THESE LINES
80 REM WILL BE
90 REM DELETED
```

Example 3 - Player/Missile (P/M) Graphics With Strings - A fast way to move player/missile graphics data is shown in this example. A dimensioned string has its string/array area offset value changed to point to the P/M graphics area. Writing to this string with an assignment statement will now write data into the P/M area at assembly language rates.

```
100 REM PLAYER/MISSILE EXAMPLE
110 DIM A$(512),B$(20)
120 X=X+1:READ A:IF A<>-l THEN B$(X,X)=CHR$(A):GOTO 120
130 DATA 0,255,129,129,129,129,129,129,129,129,255,0,-1
2000 POKE 559,62:POKE 704,88
2020 I=PEEK(106)-16:POKE 54279,I
2030 POKE 53277,3:POKE 710,224
2040 VTAB=PEEK(134)+PEEK(135)*256
2050 ATAB=PEEK(140)+PEEK(141)*256
2060 OFFS=I*256+1024-ATAB
2070 HI=INT(OFFS/256):LO=OFFS-HI*256
2090 POKE VTAB+2,LO:POKE VTAB+3,HI
3000 Y=60:Z=100:V=1:H=1
4000 A$(Y,Y+11)=B$:POKE 53248,Z
4010 Y=Y+V:Z=Z+H
4020 IF Y>213 OR Y<33 THEN V=-V
4030 IF Z>206 OR Z<49 THEN H=-H
```

```
4420 GOTO 4000
```

Chapter XI

Appendix A. Memory utilization

11 Appendix A. Memory utilization

Memory utilization with the Atari Home Computer can be difficult. Much of the memory is commandeered by the operating system, the resident cartridge, or the DOS, leaving very little under the direct control of the programmer. This gives the programmer very little freedom in laying out memory utilization for a program. With proper planning this need not be a serious problem.

PAGE ZERO

The most important RAM for any assembly language programmer is page zero. Page zero is absolutely essential for pointers and is very useful for heavily used variables, because code written for page zero variables is more compact and runs faster. Hence the assembly language programmer wants to know how many locations on page zero can be stolen for his own use. This appendix will not cover the use of page zero locations as they are defined and used by firmware. Instead, it will address only their use when the firmware's function is disabled or ignored.

The lower half of page zero (addresses $00-$80) is reserved for use by the operating system. The 128 bytes here are required for the entire array of services that the operating system provides. Most programs will use only a portion of those services. Thus, many locations in this region will not be used by the operating system during the program's execution. In particular, the 43 bytes from $50 to $7A are used only by the screen editor and display handler. Most programs will use custom display lists with their own display handlers. These 43 bytes would then become available during program execution. Similar reasoning applied to other locations dedicated to special functions would free even more.

Unfortunately, there is a major flaw in this reasoning. The software department at Atari is constantly evaluating the performance of the operating system and making changes in the code. There are now two versions of the operating system -- Rev A and Rev B. They are functionally almost identical; software that runs under Rev A will almost certainly run under Rev B. More revisions are contemplated and it is highly probable that the under utilization of those 43 bytes will be corrected in future revisions of the operating system. Thus, any software packages that steal those 43 bytes, or any other bytes from the lower half of page zero, will probably malfunction if run under a future operating system. Therefore, commercially offered software should not steal any bytes from the lower half of page zero.

The programmer must look to the upper half of page zero for free bytes. These 128 bytes are reserved for use by the cartridge. If no cartridge is in place, they are free. If a cartridge is in place, some bytes will be reserved for the programmer. The BASIC cartridge leaves only 7 bytes for the use of the programmer -- $CB through $D1. The programmer who must have more page zero bytes has only one option: the bytes used by the floating point package ($D4 to $FF). These 44 bytes may be taken if the floating point package is not used by the programmer's routines and if those routines do not themselves call the floating point package. The programmer does not have unlimited use of these bytes; they must not be used by any interrupt routines, as such routines might strike during a floating point operation called by BASIC. There are no other bytes on the upper half of page zero usable by the programmer.

The programmer working in the BASIC environment is seldom interested in the high performance and compactness that page zero offers; after all, if high speed and compactness were primary concerns, the programmer would not have chosen BASIC as a delivery language. The programmer most interested in large amounts of page zero is the assembly language programmer. A pure assembly language program does not need any cartridge installed to run; therefore it should have access to all 128 bytes of the upper half of page zero. In practice this is not quite so simple, for an assembly language program must be debugged, and the only convenient way to do this at present is with the debugger in the Atari Assembler-Editor cartridge. Assembly language programmers will note with chagrin that this cartridge only reserves 32 bytes for their own use. While this is entirely adequate for virtually any program it is not enough to sate the appetite of a high-performance program. There are 30 more bytes that are not used by the debugger portion of the cartridge. These are: $A4, $A5, $AD, $AE, $DB through $E5, $EA through $F1, $F5, $F6, $F9 through $FB, $FE, and $FF. If these bytes are used the programmer must not return to the Editor or the Assembler or bad things may happen. Furthermore, the mini-assembler inside the debugger must not be used.

If you use other cartridges or disk based languages, you are on your own. Many of these systems use even more of page zero.

ABSOLUTE RAM

Another problem the programmer faces is presented by the DOS. It is very desirable to write programs that run on either 16K cassette systems or 48K diskette systems. Unfortunately, such systems have very little free RAM in common, for the DOS and DUP would almost fill the RAM of the 16K system. There are several solutions to this problem. One is to produce two different versions of the code, a disk version and a cassette version. The two programs would be orged to different locations. This means that customers who buy the cassette version will not be able to transport it to a diskette if they upgrade their systems.

There is another way. Page six is common free RAM for all systems. Variables and vectors can be placed on page six and used by cassette-based software or diskette-based software. A directory of routine addresses can be computed and placed at run-time onto page six. Machine language routines can then jump through the page six directory to the routines elsewhere in RAM. The technique can be difficult to execute; it is not recommended for large assembly language projects. Its greatest value is with medium-size machine language routines (about 1K-2K) embedded inside BASIC programs.

Chapter XII

Appendix B. Human engineering

12 Appendix B. Human engineering

The ATARI Home Computer is first and foremost a consumer computer. The hardware was designed to make this computer easy for consumers to use. Many of the hardware features protect the consumer from inadvertent errors. Software written for this computer should reflect an equal concern for the consumer. The average consumer is unfamiliar with the conventions and traditions of the computer world. Once he understands a program he will use it well most of the time. Occasionally he will be careless and make mistakes. It is the programmer's responsibility to try to protect the consumer from his own mistakes.

The current state of software human engineering in the personal computer industry is dismal. A great many programs are being sold that contain very poor human engineering. The worst offenders are written by amateur programmers, but even software written at some of the largest firms shows occasional lapses in human engineering.

Human engineering is an art, not a science. It demands great technical skill but it also requires insight and feel. As such it is a highly subjective field devoid of absolutes. This appendix is the work of one hand, and so betrays the subjectivities of its author. A proper regard for the wide variety of opinions on the subject would have inflated this appendix beyond all reasonable limits of length. Furthermore, a complete presentation of all points of view would only confuse the reader with its many assertions, qualifications, counterpoints, and contradictions. I therefore chose the simpler task of presenting only my own point of view, giving weak lip service to the most serious objections. The result is contradictory enough to satisfy even the most academic of readers.

THE COMPUTER AS SENTIENT BEING
An instructive way of viewing the problem of human engineering is to cast the programmer as sorcerer, conjuring up an intelligent being, a homunculus, within the innards of the computer. This creature lacks physical embodiment, but possesses intellectual traits, specifically, the ability to process and organize information. The user of the program enters into a relationship with this homunculus. The two sentient beings think differently; the human's thought patterns are associative, integrated, and diffuse, while the program's thought processes are direct, analytical, and specific. These differences are complementary and productive because the homunculus can do well what the human cannot. Unfortunately, these differences also create a communications barrier between the human and the homunculus. They have so much to say to each other because they are so different, but because they are different they cannot communicate well. The central problem in good programming must therefore be to provide for better communications between the user and the homunculus. Sad to say, many programmers expend the greater part of their efforts on expanding and improving the processing power of their programs. This only produces a more intelligent being with no eyes to see and no mouth to speak.

The current crop of personal computers have attained throughputs which make them capable of sustaining programs intelligent enough to meet many of the average consumer's needs. The primary limiting factor is no longer clock speed or resident memory; the primary limiting factor is the thin pipeline connecting our now-intelligent homunculus with his human user. Each can process information rapidly and efficiently; only the narrow pipeline between them slows down the interaction.
COMMUNICATION BETWEEN HUMAN AND MACHINE
How can we widen the pipeline between the two thinkers? We must focus on the language with which they communicate. Like any language, a man-machine language is restricted by the physical means of expression available to the speakers. Because the computer and the human are physically different, their modes of expression are physically different. This forces us to create a language which is not bidirectional (as human languages are). Instead, a man- machine language will have two channels, an input channel and an output channel. Just as we study human language by first studying the sounds that the human vocal tract can generate, we begin by examining the physical components of the man-machine interface.

OUTPUT (FROM COMPUTER TO HUMAN)

There are two primary output channels from the computer to the user. The first is the television screen; the second is the television speaker. Fortunately, these are flexible devices which permit a broad range of expression. The main body of this book describes the features available from the

computer's point of view. For the purposes of this appendix, it is more useful to discuss these devices in terms of the human point of view. Of the two devices (screen and speaker) the display screen is easily the more expressive and powerful device. The human eye is a more finely developed information gathering device than the human ear. In electrical engineering terms, it has more bandwidth than the ear. The eye can process three major forms of visual information: shapes, color, and animation.

Shapes

Shapes are an ideal means for presenting information to the human. The human retina is especially adept at recognizing shapes. The most direct use of shapes is for direct depiction of objects. If you want the program to tell the user about something, draw a picture of it. A picture is direct, obvious, and immediate.

The second use of shapes is for symbols. Some concepts in the human lexicon defy direct depiction. Concepts like love, infinity, and direction cannot be shown with pictures. They must instead be conveyed with symbols, such as a heart, a horizontal figure 8, or an arrow. These are a few of the many symbols that we all recognize and use. Sometimes you can create an ad hoc symbol for limited use in your program. Most people can pick up such an ad hoc symbol quite readily. Symbols are a compact way to express an idea but they should not be used in place of pictures unless compactness is essential. A symbol is an indirect expression; a picture is a direct expression. The picture conveys the idea more forcefully.

The third and most common use of shapes is for text. A letter is a symbol; we put letters together to form words. The language we thereby produce is extremely rich in its expressive power. Truly is it said, "If you can't say it, you don't know it." This expressive power is gained at a price: extreme indirection. The word that expresses an idea has no sensory or emotional connection with the idea. The human is forced to carry out extensive mental gymnastics to decipher the word. Of course, we do it so often that we have become quite fluent at translating strings of letters into ideas. We do not notice the effort. The important point is that the indirection detracts from the immediacy and forcefulness of the communication.

There is a school of thought that maintains that text is superior to graphics for communications purposes. The gist of the argument is that text encourages freer use of the reader's rich imagination. The argument does not satisfy me, for if the reader must use his imagination, he is supplying information that is not inherent in the communication itself. An equal exercise of imagination with graphics would provide even greater results. A more compelling argument for text is that its indirection allows it to pack a considerable amount of information into a small space. The space constraints on any real communication make text's greater compactness valuable. Nevertheless, this does not make text superior to graphics; it makes text more economical. Graphics requires more space, time, memory, or money, but it also communicates better than text. To some extent, the choice between graphics and text is a matter of taste, and the taste of the buying public is beyond question. Compare the popularity of television with that of radio, or movies with books. Graphics beats text easily.

Color

Color is another vehicle for conveying information. It is less powerful than shape, and so normally plays a secondary role to shape in visual presentations. Its most frequent use is to differentiate between otherwise indistinguishable shapes. It also plays an important role in providing cues to the user. Good color can salvage an otherwise ambiguous shape. For example, a tree represented as a character must fit inside an 8x8 pixel grid. The grid is too small to draw a recognizable tree; however, by coloring the tree green, the Image becomes much easier to recognize. Color is also useful for attracting attention or signalling important material. Hot colors attract attention. Color also provides aesthetic enhancement. Colored images are more pleasing to look at than black and white images.

Animation

I use the term "animation" here to designate any visual change. Animation includes changing colors, changing shapes, moving foreground objects, or moving the background. Animation's

primary value is for showing dynamic processes. Indeed, graphic animation is the only way to successfully present highly active events. The value of animation is most forcefully demonstrated by a game like STAR RAIDERS. Can you imagine what the game would be like without animation? For that matter, can you imagine what it would be like in pure text? The value of animation extends far beyond games. Animation allows the designer to clearly show dynamic, changing events. Animation is one of the major advantages that computers have over paper as an information technology. Finally, animation is very powerful in sensory terms. The human eye is organized to respond strongly to changes in the visual field. Animation can attract the eye's attention and increase the user's involvement in the program.

Sound

Graphics images must be looked at to have effect. Sound can reach the user even when the user is not paying direct attention to the sound. Sound therefore has great value as an annunciator or warning cue. A wide variety of beeps, tones, and grunts can be used to signal feedback to the user. Correct actions can be answered with a pleasant bell tone. Incorrect actions can be answered with a raspberry. Warning conditions can be noted with a honk.

Sound has a second use: providing realistic sound effects. Quality sound effects can greatly add to the impact of a program because the sound provides a second channel of information flow that is effective even when the user is visually occupied.

Sound is ill-suited for conveying straight factual information; most people do not have the aural acuity to distinguish fine tone differences. Sound is much more effective for conveying emotional states or responses. Most people have a large array of associations of sounds with emotional states. A descending sequence of notes implies deteriorating circumstances. An explosion sound denotes destruction. A fanfare announces an important arrival. Certain note sequences from widely recognized popular songs are immediately associated with particular feelings. For example, in ENERGY CZAR, a funeral dirge tells the user that his energy mismanagement had ruined America's energy situation, and a fragment of "Happy Days Are Here Again" indicates success.

INPUT DEVICES (FROM HUMAN TO COMPUTER)

There are three input devices most commonly used with the ATARI Home Computer. These are the keyboard, joystick, and paddles.

Keyboard

The keyboard is easily the most powerful input device available to the designer. It has over 50 direct keystrokes immediately available. Use of the CONTROL and SHIFT keys more than doubles the number of distinguishable entries the user can make. The CAPS/LOWR and ATARI keys extend the expressive range of the keyboard even further. Thus, with a single keystroke the user can-designate one of 125 commands. A pair of keystrokes can address more than 15,000 selections. Obviously, this device is very expressive; it can easily handle the communications needs of any program. For this reason the keyboard is the input device of choice among programmers.

While the strengths of the keyboard are undeniable, its weaknesses are seldom recognized. Its first weakness is that not many people know how to use it well. Programmers use keyboards heavily in their daily work; consequently, they are fast typists. The average consumer is not so comfortable with a keyboard. He can easily press the wrong key. The very existence of all those keys and the knowledge that one must press the correct key is itself intimidating to most people.

A second weakness of the keyboard is its indirection. It is very hard to attach direct meaning to a keyboard. A keyboard has no obvious emotional or sensory significance. The new user has great difficulty linking to it. All work with the keyboard is symbolic, using buttons which are marked with symbols which are assigned meaning by the circumstances. The indirection of it all can be most confusing to the beginner. Keyboards also suffer from their natural association with text displays; I have already discussed the weaknesses of text as a medium for information transfer.

Another property of the keyboard that the designer must keep in mind is its digital nature. The

keyboard is digital both in selection and in time. This provides some protection against errors. Because keystroke reading over time is not continous but digital, the keyboard is not well-suited to real-time applications. Since humans are real-time creatures, this is a weakness. The designer must realize that use of the keyboard will nudge him away from real-time interaction with his target user.

Paddles

Paddles are the only truly analog input devices readily available for the system. As such they suffer from the standard problem all analog input devices share: the requirement that the user make precise settings to get a result. Their angular resolution is poor, and thermal effects produce some jitter in even an untouched paddle's output.

Their primary value is twofold. First, they are well-suited for choosing values of a one-dimensional variable. People can immediately pick up the idea that the paddle sweeps through all values, and pressing the trigger makes the selection known. Second, the user can sweep from one end of the spectrum to the other with a twist of the dial. This makes the entire spectrum of values immediately accessible to the user.

An important factor in the use of paddles is the creation of a closed input/output loop. In most input processes, it is desirable to echo inputs to the screen so that the user can verify the input he has entered. This echoing process creates a closed input/output loop. Information travels from the user to the input device to the computer to the screen to the user. Because the paddle has no absolute positions, echoing is essential.

Any set of inputs that can be meaningfully placed along a linear sequence can be addressed with a paddle. For example, menus can be addressed with a paddle. The sequence is from the top of the menu to the bottom. it is quite possible (but entirely unreasonable) to substitute a paddle for a keyboard. The paddle sweeps through the letters of the alphabet, with the current letter being addressed shown on the screen. Pressing the paddle trigger selects the letter. While the scheme would not produce any typing speed records, it is useful for children and the idea could be applied to other problems.

Joysticks

Joysticks are the simplest input devices available for the computer. They are very sturdy and so can be used in harsh environments. They contain only five switches. For this reason their expressive power is frequently underestimated. However, joysticks are surprisingly useful input devices. When used with a cursor, a joystick can address any point on the screen, making a selection with the red button. With proper screen layout, the joystick can thus provide a wide variety of control functions. I have used a joystick to control a nuclear reactor (SCRAM) and run a wargame (EASTERN FRONT 1941).

The key to the proper use of the joystick is the realization that the critical variable is not the selection of a switch, but the duration of time for which the switch is pressed. By controlling how long the switch is pressed, the user determines how far the cursor moves. This normally requires a constant velocity cursor. A constant velocity cursor introduces a difficult trade-off. If the cursor moves too fast, the user will have difficulty positioning it on the item of choice. If the cursor moves too slowly, the user will become impatient waiting for it to traverse long screen distances. One solution to this problem is the accelerating cursor. If the cursor starts moving slowly and accelerates, the user can have both fine positioning and high speed.

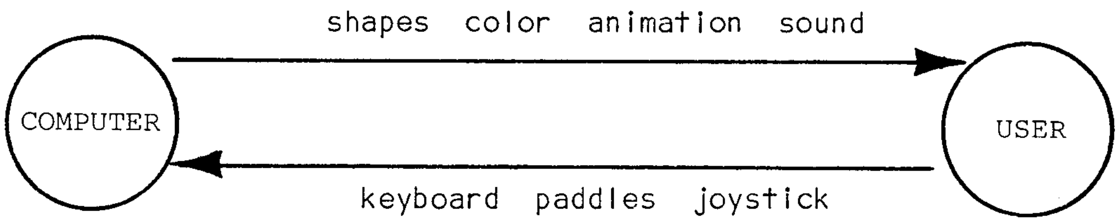
The real value of the joystick is its high tactility. The joystick involves the user in his inputs in a direct and sensory way. The tactility of the keyboard is not emotionally significant. A joystick makes sense --- push up to go up, down to go down. If the cursor reflects this on the screen, the entire input process makes much more sense to the user.

Joysticks have their limitations. Although it is possible to press the joystick in a diagonal direction and get a correct reading of the direction, the directions are not distinct enough to allow diagonal entries as separate commands. Just as some words (e.g., "library," "February") are hard to enunciate clearly, so too are diagonal orders hard to enter distinctly. Thus, diagonal values should

be avoided unless they are used in the pure geometrical sense: up on the joystick means up, right means right, and diagonally means diagonally.

SUMMARY OF COMMUNICATIONS ELEMENTS

We have discussed a number of features and devices which, taken together, constitute the elements of a language for interaction between the computer and the user. They are:



CONSTRUCTING A LANGUAGE
How do we assemble all of these elements into an effective language? To do so, we must first determine the major traits we expect of a good language:

- o Completeness
- o Directness
- o Closure
- o Completeness

The language must completely express all of the ideas that need to be communicated between the computer and the user. It need not express ideas internal to either thinker's thought processes. For example, the language used in STAR RAIDERS must express all concepts related to the control of the vessel and the combat situation. It need not express the player's anxiety or the flight path intentions of the Zylons. These concepts, while very germane to the entire game function, need not be communicated between user and computer.

Completeness is an obvious function of any language, one that all programmers recognize intuitively. Problems with completeness most often arise when the programmer must add functions to the program, functions which cannot be supported by the language the programmer has created. This can be quite exasperating, for in many cases the additional functions are easily implemented in the program itself. The limiting factor is always the difficulty of adding new expressions to the I/O language.

Directness

Any new language is hard to learn. No user has time to waste in learning an unnecessarily florid language. The language a programmer creates for a program must be direct and to the point. It must rely as much as possible on communications conventions that the user already knows. It must be emotionally direct and obvious. For example, a CONTROL-X keystroke is obscure. What does it mean? Perhaps it means that something should be destroyed; X implies elimination or negation. Perhaps it implies that something should be examined, expunged, exhumed, or something similar. if none of these possibilities are indeed the case, then the command is unacceptably indirect. Keyboards are notorious for creating this kind of problem.

Closure

Closure is the aspect of communications design that causes the greatest problems. The concept is best explained with an analogy. The user is at point A and wishes to use the program to get to point B. A poorly human-engineered program is like a tightrope stretched between points A and B. The user who knows exactly what to do and performs perfectly will succeed. More likely, he or she will slip and fall. Some programs try to help by providing a manual or internal warnings that tell the user what to do and what not to do. These are analogous to signs along the tightrope advising "BE

CAREFUL" and "DON'T FALL." I have seen several programs that place signs underneath the tightrope, so that the user can at least see why he failed as he plummets. A somewhat better class of programs provide masks against illegal entries. These are equivalent to guardrails alongside the tightrope. These are much nicer, but they must be very well constructed to ensure that the user does not thwart them. Some programs have nasty messages that bark at the errant user, warning against making certain entries. These are analogous to scowling monitors in the school halls, and are useful only for making an adult feel like a child. The ideal program is like a tunnel bored through solid rock. There is but one path, the path leading to success. The user has no options but to succeed.

The essence of closure is the narrowing of options, the elimination of possibilities, the placement of rock solid walls around the user. Good design is not an accumulative process of piling lots of features onto a basic architecture; good design requires the programmer to strip away minor features, petty options, and general trivia.

This thesis clashes with the values of many programmers. Programmers crave complete freedom to exercise power over the computer. Their most common complaint against a program is that it somehow restricts their options. Thus, deliberate advocacy of closure is met with shocked incredulity. Why would anyone be so foolish as to restrict the power of this wonderful tool?

The answer lies in the difference between the consumer and the programmer. The programmer devotes his life to the computer; the consumer is a casual acquaintance at best. The programmer uses the computer so heavily that it is cost-effective to take the time to learn to use a more powerful tool. The consumer does not have the time to lavish on the machine. He wants to get to point B as quickly as possible. He does not care for the fine points that occupy a programmer's life. Bells and whistles cherished by programmers are only trivia to him. You as a programmer may not share the consumer's values, but if you want to maintain your livelihood you had better cater to them.

Closure is obtained by creating inputs and outputs that do not admit illegal values. This is extremely difficult to do with a keyboard, for a keyboard always allows more entries than any real program would need. This is an excellent argument against the use of the keyboard. A joystick is much better, because you can do so little with it. Because it can do so little, it is easier to conceptually exclude bad inputs. The ideal is achieved when all necessary options are expressible with the joystick, and no further options will fit. In this case the user cannot make a bad entry because it doesn't exist. More important, like Newspeak in Orwell's "1984", the user cannot even conceive bad thoughts because no words (inputs) for them even exist.

Closure is much more than masking out bad inputs. Masking makes bad inputs conceivable and expressible, but not functional. For example, a keyboard might be used with the "M" key disabled because it is meaningless. The user can still see the key, he can imagine pressing it, and he can wonder what would happen if he did press it --- all wasted effort. The user can waste even more time by pressing it and wondering why nothing happened. The waste is compounded by the programmer imagining the user doing all these wasteful things and putting in code to stop the symptoms without eliminating the disease. By contrast, a properly closed input structure uses an input device which can express only the entries necessary to run the program, and nothing more. The user can't waste time messing with something that isn't there.

The advantages that accrue when closure is properly applied are manifold. Code is tighter and runs faster because there need be no input error checking; such errors are obsolete in the new program. The user requires less time to learn the program and has fewer problems with it.

The primary problem with closure is the design effort that must be expended to achieve good closure. The entire relationship between the user and the program must be carefully analysed to determine the minimum vocabulary necessary for the two to communicate. Numerous schemes of communication must be examined and discarded before the true minimum scheme is found. In the process, many bells and whistles that the programmer wanted to add will have to be eliminated. If the programmer objectively looks beyond his own values, he will often conclude that the bells and whistles are more clutter than chrome.

CONCLUSIONS

The design of the language of communication between the user and the program will be the most difficult part of the design process in consumer software. The designer must carefully weigh the capabilities of the machine and the needs of the user. He must precisely define the information that must flow between the two sentient beings. He must then design his language to maximize the clarity (not the quantity) of information flowing to the user while minimizing the effort the user must expend to communicate with the computer. His language must utilize the machine's features and devices effectively while maintaining its own completeness, directness, and closure.
SOME COMMON PROBLEMS IN HUMAN ENGINEERING
Having discussed the problems of human engineering in theoretical terms, we now turn to discuss specific application problems in human engineering. The list of problems is not exhaustive; it merely covers some of the problems common to almost all programs.

DELAY TIMES

Many programs require extensive computations. Indeed, almost all programs execute at some time computations that take more than a few seconds to perform. What does the user experience while these computations are executed? Too many programs simply stop the dialogue with the user for the duration of the computation. The user is left with an inactive screen and no sign of life from the computer. The computer does not respond to the user's inputs. If human engineering is created by the language of communication between the computer and the user, then this complete absence of communication can only be regarded as a total lack of human engineering. Leaving the user in the lurch like this is absolutely unforgivable.

Separate Processes

The best way to deal with the problem of reconciling computations with attentiveness is to separate the input process from the computational process. The user should be able to make inputs while the computations are proceeding. This is technically achievable; by using vertical blank interrupts the programmer can multitask input processing with mainline processing. The technique is used in EASTERN FRONT 1941. The real problem with the technique is that many problems are intrinsically sequential in nature. It is essential that the user input a value or choice before the computation can proceed to the next step. This makes it difficult to separate input processing from the mainline processing. However it is possible with clever design to perform anticipatory calculations that will determine Intermediate values so that as soon as the critical data is entered, the result might be more quickly obtained. Application of such techniques can surely reduce the delay times that the user experiences.

Speed up the Program

Another means of dealing with this problem is to speed up the program itself. Critical code can often be rewritten to decrease execution time. Proper nesting of loops (the loop with more iterations should be inside the loop with fewer iterations) can reduce execution time. Careful attention to the details of execution can yield further time reductions. Major gains can be made by converting BASIC to assembly language. Assembly is from 10 to 1000 times faster than BASIC. Assembly's advantage is greatest for memory move routines and graphics and least for floating point calculations. By masking out vertical blank interrupts, more 6502 execution time can be freed for mainline processing. Other gains can be accomplished by reducing the DMA overhead ANTIC imposes. This can be done by going to a simple graphics mode (BASIC mode 3 is best). Shortening the display list is another way to reduce DMA costs. Turning off ANTIC altogether is a drastic route which only creates the additional problem of presenting the user with a blank screen.

Entertain the User

The third way to deal with delay times is to occupy the user during the computation. A countdown is one such method. The user sees a countdown on the screen. When the countdown reaches zero, the program is back in business. Another way is to draw random graphics on the screen. The delay period should always start with a courteous message advising the user of the delay. It should also be terminated with a bell or other annunciator. You should not expect the user to keep his eyes on the screen for an arbitrary period of time. Entertaining the user during delays is a poor way to deal with delays that shouldn't have been there in the first place, but it's better than abandoning

the user.

DEALING WITH BAD USER INPUTS

The most serious problem with present consumer software is the inadequate way that bad user inputs are handled. Good designs preclude this problem by providing input languages that do not make any bad entries available. As I pointed out earlier, this is most easily accomplished with a joystick. However, there are applications (primarily text-intensive ones) that require a keyboard. Furthermore, even joysticks occasionally introduce problems with user input. How are such bad inputs to be dealt with when they cannot be expunged? Several suggestions follow. It is imperative that any protection system be applied uniformly throughout the entire program. Once the user encounters protection, he will expect it in all cases. The lack of such protection creates a gap through which the user, thinking himself secure, will surely plunge.

Flag the Error and Suggest Solution

The most desirable approach in this unpleasant situation is to flag the user's error on the screen in plain language and suggest a correct entry. Three things must be included in the computer's response. First, the user's entry must be echoed back so he knows what he did that caused the problem. Second, the offending component of the entry must be clearly marked and explained so that the user knows why it is wrong. Third, an alternate legal entry must be suggested so that the user does not become frustrated by the feeling that he has encountered a brick wall. For example, an appropriate response to a bad keystroke entry might read thusly: "You pressed CONTROL-A, which is an autopsy request. I cannot perform autopsies on living people. I suggest you kill the subject first."

This method is obviously very expensive in terms of program size and programming time. That is the price one pays for bad design. There are less expensive and less effective methods.

Masking out Bad Keys

One common solution to keyboard input problems is to mask out all bad entries. If the user presses a bad key, nothing happens. No keyboard click is generated and no character appears on the screen. The program only hears what it wants to hear. This solution is secure in that it prevents program crashes, but it does not protect the user from confusion. The user would only press a key if he felt that it would do something for him. Masking out the key cannot correct the user's mistaken impression. It can only lead him to the conclusion that something is seriously wrong with his computer. We don't want to do this to our users.

A variant on this scheme is to add a nasty buzzer or raspberry to chastise the user for his foolishness. Indeed, some amateurish programs go so far as to heap textual abuse on the user. Such techniques are highly questionable. There may indeed be cases requiring dangerous keystroke entries which are guarded by fierce and nasty messages; such cases are quite rare. Corrective messages should always conform to high standards of civility.

Error Messages

An even cheaper solution is to simply post an error message on the screen. The user is told only that he did something wrong. In many cases, the error message is cryptic and does not help the user in the least. ATARI BASIC is an extreme example of this. Error messages are provided by number only. This can be justified only when the program must operate under very tight memory constraints.

In most cases, the designer chooses to sacrifice human engineering features such as meaningful error messages for some additional technical power. As pointed out in the beginning of this appendix, we are reaching the stage in which additional technical power is no longer a limiting factor to consumers, but human engineering is a limiting factor. Thus, the trade-off is less justifiable.

Protection/Power Trade-Offs

One objection to many human engineering features is that they slow down the user's interaction with the computer. Programmers tire of Incessant "ARE YOU SURE?" requests and similar restrictions. One solution to this problem is to provide variable protection/power ratios. For example, a program can default to a highly protected state on initialization. All entries are carefully checked and echoed to the user for confirmation. The user has an option to shed protection and work in high-speed mode. This option is not obvious from the screen --- it is only described in the documentation. Thus, the intensive user can work at a fast pace and the casual user can have adequate protection.

MENUS AND SELECTION TECHNIQUES

Menus are standard devices for making the user aware of the options available. They are especially useful for beginning users. Command-oriented schemes preferred by programmers confuse beginners who cannot afford the time investment to learn the lexicon of commands used by a command-oriented program. We will discuss several common problems associated with the use of menus.

Menu Size

How many entries should be on a menu? The obvious upper limit is dictated by the size of the screen, but this limit is too large, for a BASIC mode 0 screen could hold up to 48 entries (24 lines with two choices per line). My guess is that seven entries is the desired upper limit on menu size. This allows plenty of screen space to separate the entries, provide a menu title, and some sort of prompt.

Multiple Menus

Frequently a program will require several menus to fully cover all of the options it offers. It is very important that multiple menus be organized in a clear manner. The user can easily get lost wandering around through such menu mazes. One way is to have a main menu that is prominently marked as such, and provide each secondary menu with an option to return to the main menu. Another way is to nest menus in a hierarchical structure. When using such methods, the programmer must provide color and sound cues to help the user ascertain his position in the menu structure. Each menu or menu level should have a distinctive note or color assigned to it. The note frequency should be associated with the position in the hierarchy.

Selection Methods

Once the user has seen his options, how does he make his choice known to the computer? The most common way is to label each entry on the menu with a letter or number; the user makes his selection by pressing the corresponding key on the keyboard. This is a clumsy solution involving unnecessary indirection. There are a number of better methods. Most of them use the same basic scheme: a movable pointer addresses an option, and a trigger selects it. One scheme highlights the option being addressed in inverse video. The SELECT button changes the pointer to address the next menu selection, with full wraparound from the end of the menu to the beginning. The START button engages a menu option. Another program automatically rotated the pointer through the menu options; the user need only push a button at the correct moment when his desired option was being addressed (not an impressive method). Paddles and joysticks are very well suited for menu selection. Either one can be used to sweep the pointer through the menu selections, with the red trigger button making the selection. My pet scheme for menu selection uses a cursor on a large scrolling menu. The user moves the cursor with a joystick. Signposts can direct her to different regions of the menu. The user makes a selection by placing the cursor directly on top of an option and pressing the trigger button.

MANUALS VERSUS ON-BOARD TEXT

A common problem with menus, error messages, prompts, and other messages is that such material can easily consume a large amount of memory --- memory that could well be used for other features. Such material could be placed in a reference document, but doing so would detract from the quality of the program's human engineering. The designer must decide how much material should go into the program and how much should be relegated to the manual. With disk-based programs it is possible to store some of the material on the diskette; this lessens the harshness of the trade-off. When the problem is approached only from the human engineering point of view, it is easily answered: all material should be included In the program, or at least on a

diskette. Economic and technical considerations argue against this. It is my personal view that each technology should be used for the things it does best. While the computer can handle static text, its forte is dynamic information processing. Paper and ink handle static information more cheaply and often more clearly than a computer. I therefore prefer to put static information into a manual and let the program refer the user to the manual. I still include critical information within the program; my dividing line bends with local needs.

MEASURES OF SUCCESS

How can a designer determine the success of his human engineering? There are several indicators that provide valuable feedback. The first is the minimum length of the manual. If you exclude background material and isolate only the material in the manual that is absolutely necessary to describe how to use the program, then the length of this material is a good measure of your human engineering. The more material, the worse you've done. A well-designed program should require very little explanation. This should not be construed as an argument against proper documentation. Documentation should always describe the program in more detail than is absolutely necessary. A long, lavish manual is good; a program that demands such a manual is not.

Another measure is the amount of time that a first-time user expends to learn to use the program satisfactorily. Good programs can be used in a matter of minutes.

A third measure is the amount of thinking a user must do to use the program. A well-designed program should require no cognitive effort to use. This does not mean that the user does not think at all while using such a program. Rather, he thinks about the content of the program rather than the mechanics of the program. He should concentrate on what he is doing, not how he does it.

The well-engineered program eliminates mental distance between the user and the computer. The two thinking beings achieve a mental syntony, an intellectual communion.

Chapter XIII

Appendix C. The Atari Cassette

13 Appendix C. The Atari Cassette

This discussion of the ATARI 410 Program Recorder includes the following topics:

* How the cassette works. Information on the hardware and software used to operate the cassette.
* Cassette applications. How to mix audio and digital information to produce a user-oriented program.

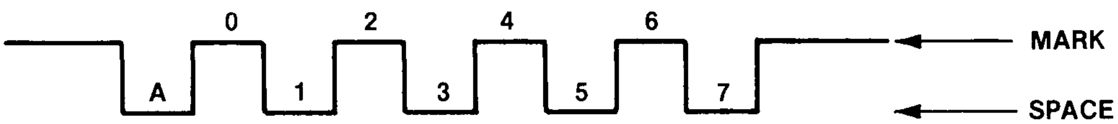
HOW THE CASSETTE WORKS

RECORD STRUCTURE

Byte Definition

The OS writes files in fixed-length blocks at 600 baud (physical bits/ second). Asynchronous serial transmission is used to read and write data between the ATARI 400/800 Computers and the ATARI Program Recorder. POKEY recognizes each data byte in this order: 1 start bit (space), 8 data bits (0=space, 1=mark), then one stop bit (mark). A byte is sent/received least significant bit first.

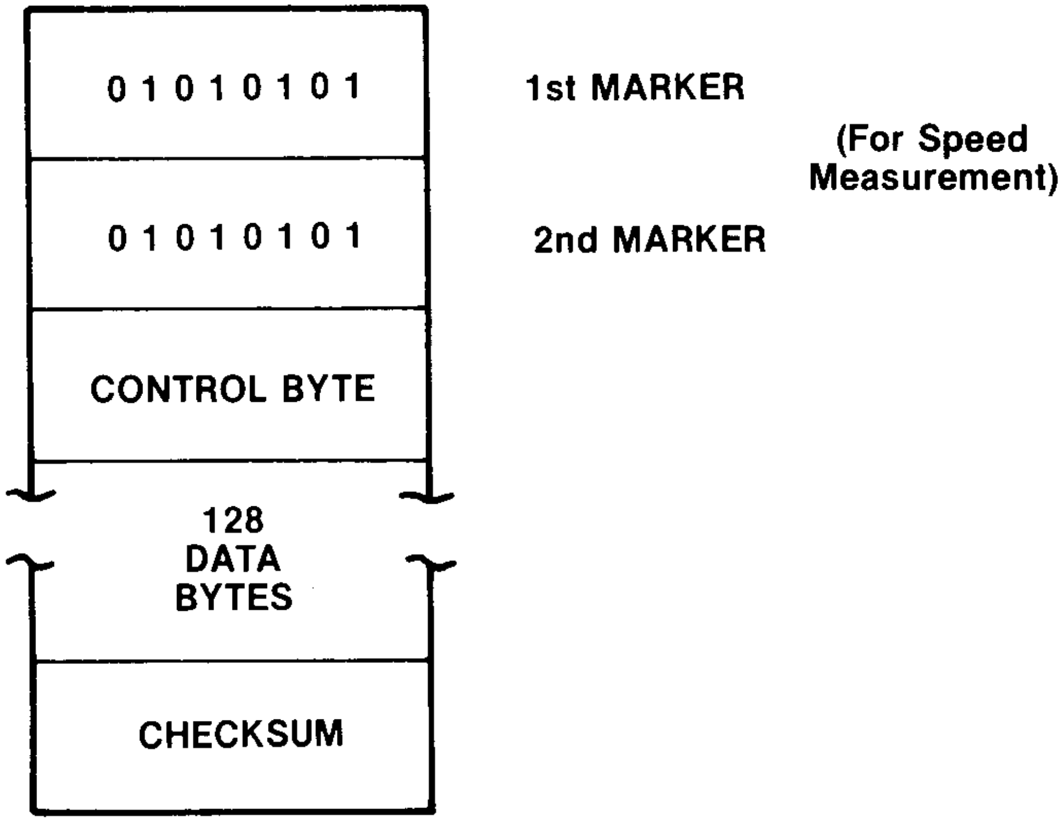
The frequency used to represent a mark is 5327 Hz. For a space, the frequency is 3995 Hz. The data byte format is as follows:



A = Start Bit (Space)
0-7 = Data Bits
B = Stop Bit (Mark)

Record Definition

Records are 132 bytes long. A record is broken down in the following way: 2 marker characters for speed measurement, a control byte, 128 data bytes, and the checksum byte. The record format is shown below:



1st and 2nd MARKER

Each marker character is a 55 (hex). Including start and stop bits, each marker is 10 bits long. Ideally, there should be no blank tape between the markers and the subsequent data.

Speed Measurement:

The purpose of the marker characters is to adjust the baud rate.

The input baud rate is assumed to be a nominal 600 baud. This is adjusted, however, by the SIO routine to account for drive motor variations, stretched tape, etc. Once the true receive baud rate is calculated, the hardware is adjusted accordingly. Input baud rates ranging from 318 to 1407 baud can theoretically be handled using this technique.

The OS checks the tape speed in the following manner: The software looks at the POKEY Serial-In bit continuously. Looking for a start (0 bit) which signifies the beginning of a record. When it finds one, the OS stores the current frame counter by saving the ANTIC VCOUNT (vertical screen counter). Continuing to look directly at the Serial-In bit, the OS counts the 20 bits (end of the 2 markers), then uses VCOUNT and the frame counter to determine the elapsed time. The baud rate to use is derived from the result. This is done for each record.

Control Byte

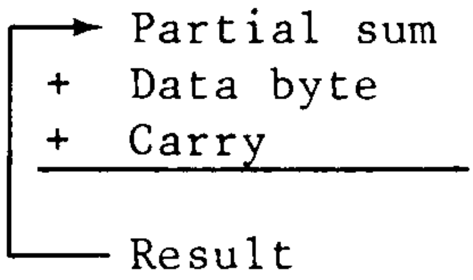
The control byte contains one of three values:

o $FC indicates the record is a full data record (128 bytes).
o $FA indicates the record is a partially full data record; fewer than 128 bytes were supplied by the user. This case may occur only in the record prior to the end-of-file. The actual number of data bytes, 1 to 127, is stored in the last data byte prior to the checksum; i.e., the 128th data byte.
o $FE indicates the record is an end-of-file record and is followed by 128 zero bytes.

Checksum

The checksum is generated and checked by the SIO routine, but is not contained in the cassette handler's I/O buffer CASBUF [03FD]

The checksum is a single byte sum of all the other bytes in the record, including the two markers. The checksum is computed with endaround carry. As each byte is added into the sum, the carry bit is also added in.



TIMING

Inter-Record Gap

As mentioned earlier, each record consists of 132 data bytes including the checksum byte. In order to distinguish one record from another, the cassette handler adds a Pre-Record Write Tone (PRWT) and Post- Record Gap (PRG). PRWT and PRG are both pure mark tone. The InterRecord Gap (IRG) between any two records thus consists of the PRG of the first record followed by the PRWT of the second record. The layout of the records and gaps is as follows:

PRWT	MARKER	DATA	PRG	PRWT	MARKER	DATA	PRG
Record 1				Record 2			

Normal IRG Mode and Short IRG Mode

The length of PRWT and PRG are dependent upon the Write Open mode. There are two types of IRG modes: Normal IRG mode and Short IRG mode.

When a file is opened, the most significant bit of AUX2 specifies the mode. On subsequent output or input, the cassette handler executes the READ/WRITE in either mode based on the MSB of the AUX2 byte:

AUX2	7	6	5	4	3	2	1	0
	C							

c = 1 indicates that the cassette is to be read/written in Short IRG mode (continuous mode).
c = 0 indicates Normal IRG mode.

Normal IRG Mode

This mode is used for a READ interleaved with processing; i.e., the tape always comes to a stop after each record is read. If the computer "STOPS" the tape and gets its processing done fast enough, the next READ may occur so quickly that the cassette deck may see only a slight dip in the control line.

Short IRG Mode

In this mode the tape is not stopped between records, either when being written or during readback.

On readback, the program must issue a READ for each record before it passes the read head. The only common use of this mode so far is storage of BASIC programs in internal (tokenized) form where, on readback, BASIC has nothing more to do with the data than put it in RAM. The special BASIC commands "CSAVE" and "CLOAD" specify this mode.

There can be a potential problem with this. The software that writes the tape must allow long enough gaps, so the beginning of records are not missed on readback.

Timing Structure

The timings for each of the inter-record gaps are as follows:

NORMAL IRG PRWT	3 seconds of mark tone.
SHORT IRG PRWT	0.25 second of mark tone.
NORMAL IRG PRG	Up to 1 second of unknown tones.
SHORT IRG PRG	From 0 to N seconds of unknown tones, where N is dependent upon user program timing.

Each record is written with the following timing: once the motor starts and the PRWT is written, the duration of the tone depends on the above format. The record follows, then the PRG is written. The motor is then stopped for Normal mode, but continues writing mark for Short IRG mode.

Note that for the Normal IRG mode, the tape will contain a section of unknown data because of stopping and restarting the motor. (Up to 1 second of travel is possible, depending on the cassette machine.) This unknown data may be garbage data left previously on the tape.

Noisy I/O Feature

The Noisy I/O feature is useful for determining the success of reading the tape, particularly with CLOAD. Marks and spaces use different sound frequencies and you quickly learn the good and bad sounds the OS makes.

FILE STRUCTURE

A file consists of the following three elements:

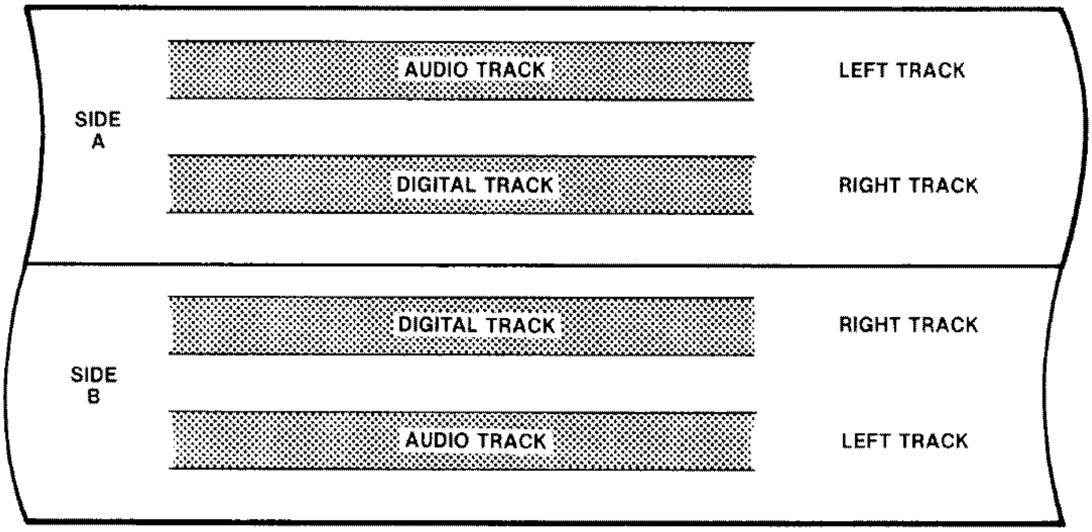
o A 20-second leader of the mark tone
o Any number of data records
o End-of-File

When the file is opened (output), the OS starts by writing a mark leader of 20 seconds, the OS then returns to the caller, but leaves the tape running and writing marks.

The WRITE/READ timeout counter is set for about 35 seconds as the OS returns. If the timeout occurs before the first record is written, the tape will stop, leaving a gap between the open leader and the first record leader.

TAPE STRUCTURE

There are two sides to each tape. Each side has two tracks, one for audio and the other one for digital recording. This way the tape can be recorded in both directions. Following is a flat view of the tape:



Tapes are recorded in 1/4 track stereo format at 1 7/8 inches per second (IPS). Note that the ATARI 800 computer utilizes a tape deck that has a stereo head configuration (not a single or mono type).

CASSETTE BOOT

The Cassette Boot program can be booted from the cassette at power-up time as part of the system initialization. System initialization performs functions such as zeroing all of the hardware registers, clearing RAM, setting flags and so on.

After all the resident handlers are brought in, if the [START] key is pressed, the Cassette Boot request flag CKEY [004A] is set. If the Cassette Boot request flag is set, then a Cassette Boot operation is attempted.

The following requirements must be met in order to boot from the cassette:

1. The operator must press the [START] key as power is applied to the system.

2. A cassette tape with a proper boot format file must be installed in the cassette drive, and the PLAY button on the recorder must be pressed.

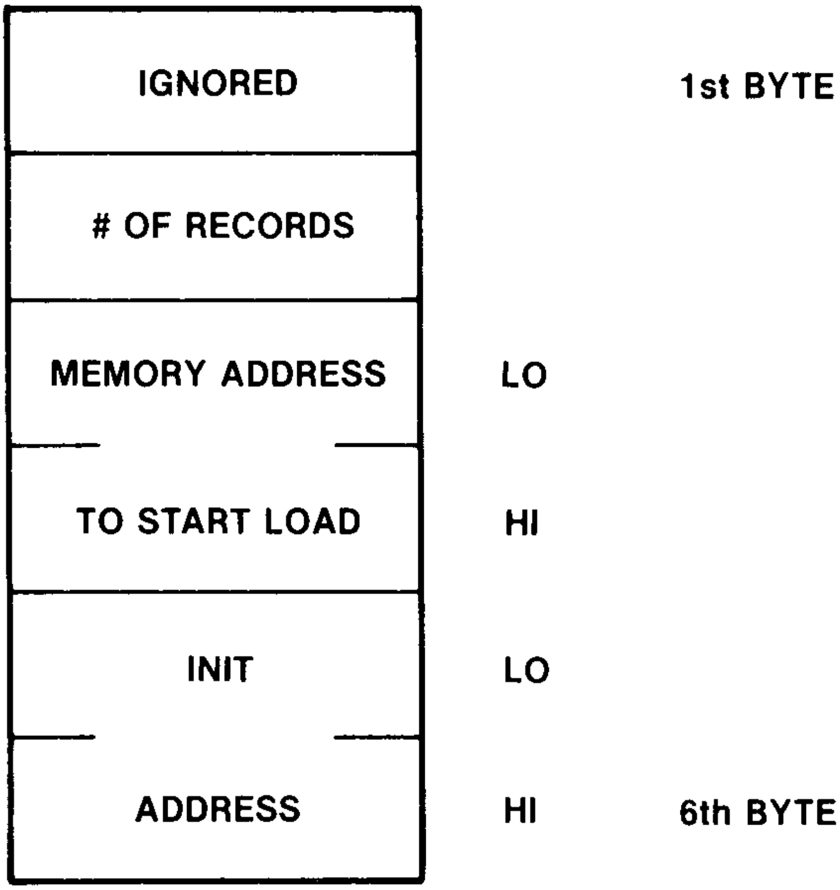
3. The cassette file must have been created in Short IRG mode.

4. When the audio prompt occurs, the operator must press a key on the keyboard.

If all of these conditions are met, the OS will READ the boot file from the cassette and then transfer control to the software that was read in. The Cassette Boot process is given in more detail below.

1. READ the first cassette record to the cassette buffer.

2. Extract information from the first 6 bytes. The first 6 bytes of a Cassette Boot file are formatted as shown below:



1ST BYTE is not used by the Cassette Boot process.

2ND BYTE contains the number of 128 byte cassette records to be read as part of the boot process (including the record containing this information). This number may range from 1 to 255, with 0 meaning 256.

3RD and 4TH BYTES contain the address (LO,HI) at which to start loading the first byte of the file.

5TH and 6TH BYTES contain the address (LO,HI) to which control is transferred after the boot process is complete. Pressing the [S/RESET] key will also transfer control to this address assuming that the boot process is complete.

When step 2 is complete, the Cassette Boot program will have:

- o Saved number of records to boot
- o Saved the load address
- o Saved the initialization address in CASINI [02,03]

3. Move the record just read to the load address specified.

4. READ the remaining records directly to the load area.

5. JSR to the load address +6 where a multistage boot process may continue. The carry bit will indicate the success of the operation (carry set = error, carry reset = success) on return.

6. JSR indirectly through CASINI for initialization of the application. The application should put its starting address into DOSVEC [0A,0B] during initialization, and then return.

7. JMP indirectly through DOSVEC to transfer control to the application.

Pressing the [S/RESET] key after the application is fully booted will cause steps 6 and 7 to repeat.

CASSETTE APPLICATIONS
HOW TO CONFIGURE THE CASSETTE SYSTEM

Most serial bus devices have two identical connectors: one is a serial bus input and the other a serial bus extender. Using these connectors, peripherals may be "daisychained" simply be cabling them together in a sequential fashion like the following diagram:

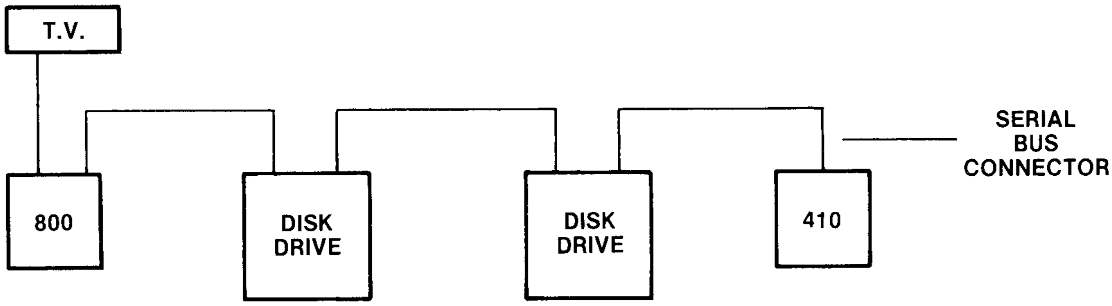


Figure C-1 Daisychained Peripheral Equipment

However, the cassette does not conform to the protocol of the other peripherals that use the serial bus. The cassette must be the last device on the serial bus because it does not have a serial bus extender connector as the other peripherals do. The lack of a bus extender assures that there is never more than one cassette drive connected to the system. The system cannot sense the absence or presence of the cassette drive, so it may be connected and disconnected at will.

Whenever there is a need to open a cassette file for reading or writing, use the following instructions:

Input (Data From 410 to 800) When the cassette is opened for input, a single audible tone is generated using the keyboard speaker. If the cassette is ready (power on, serial bus cable connected, tape cued to start of file), the user must depress the PLAY button on the cassette and any ATARI 800 keyboard key (except [BREAK]) to initiate tape reading.

OUTPUT (DATA FROM 800 to 410) When the cassette is opened for output, two separate audible tones are generated using the keyboard speaker. If the cassette is ready (as previously described), the user must simultaneously press the PLAY and RECORD buttons on the cassette, and then press any keyboard key (except [BREAK]) to initiate writing the tape.

SAVING AND LOADING DIGITAL PROGRAMS

Concept

The following technique saves the digital data directly from the computer through its I/O port of either the ATARI 410 Program Recorder or the Atari Lab Machine which uses 1/4 inch tape recorded at 7 1/2 inches per second.

FOR BASIC:

Format: CSAVE
100 CSAVE
This command is usually used in direct mode to save a RAM-resident program onto cassette tape. CSAVE writes the tokenized version of the program to the 410.

Format: CLOAD
100 CLOAD
This command can be used in either direct or deferred mode to read programs from cassette tape into RAM for execution.

FOR ASSEMBLY LANGUAGE:

Source Program

Format: LIST#C:[,XX,YY]

This command is used to write assembly source code. The items in the optional brackets [,XX,YY] mean to transfer only lines XX to YY to cassette. If line numbers are not provided, the whole program is listed to cassette.

Format: ENTER#C:

This command reads source code from the cassette.

Object Program

Format: SAVE#C:

The contents of a block of memory, locations XXXX to YYYY, is saved onto cassette.

Format: LOAD#C:

This command will load memory with the material that was previously saved. The range of memory locations that are filled will be the same as those given in the original SAVE command.

SAVING DIGITAL PROGRAMS WITH AUDIO AS BACKGROUND

Concept

This recording technique does not allow any program control over the audio. The audio plays purely as background to help time pass during the monotonous loading process.

Step 1: Follow the digital writing instructions indicated in 2.2 for BASIC and assembly programs; except, this time ATARI standard cassette tape (1 7/8 inches per second) is not used. Because it is hard later for an individual to record audio onto the program recorder, we have to use the ATARI recording lab machine, which uses 7 1/2 inches per second master tape. The lab machine is a much more sophisticated recording machine able to record data onto a specified track.

On the lab machine, the recording mode is switched to ON for the right track, so digital is saved onto the right track of the 7 1/2 inch tape.

Step 2: Use Step 1 for audio recording, but first rewind the tape to the beginning of the program then switch the recording mode to ON for left track. This way the audio is recorded onto the left track of the 7 1/2 inch tape.

DIGITAL PROGRAMS, AUDIO, SYNC MARK, AND SCREEN MANAGEMENT

Sync Mark Concept

There is no efficient way for the program to detect an audio segment when the cassette is playing. To solve the synchronization problem, Sync Mark is used to carry the signal to inform the program that an audio segment has been played (an audio segment can be either a piece of music or an instruction, depending on the application).

More precisely, since audio data has no record structure, Sync Mark recorded on the digital track is more or less like End-of-Record Mark for audio. For example, once the program senses the Sync Mark, the program can decide what to do next, like stop the cassette motor for lengthy processing or continue to play the next audio segment.

Step 1: The programmer figures out an audio script for FROG. The script is like this:

(MUSIC) TODAY I AM GOING TO TELL YOU A FAIRY-TALE NAMED "THE PRINCESS AND THE FROG." IT IS A SWEET STORY SO DON'T GO AWAY. /

(MUSIC) BEFORE I START MY STORY, I WOULD LIKE TO KNOW WHO I AM TALKING TO. WHAT IS YOUR NAME? TYPE YOUR NAME AND PRESS CARRIAGE RETURN. (PAUSE)

(MUSIC) NOW, LET'S START THE STORY. ONCE UPON A TIME, THERE WAS THIS BEAUTIFUL PRINCESS LIVING IN A CASTLE AND HER NAME WAS YYYY. /

(MUSIC) ON A CLEAR AND BEAUTIFUL DAY, THE PRINCESS WAS WALKING ALONG THE .... /

REMARK:
- "/" means the program is checking for a sync mark. It is best if the speaker pauses about 1/2 second here before continuing to the next segment of the audio script.
- "PAUSE" is to indicate that the speaker pauses about 1 second here to allow time for the stopping and starting of the cassette motor. Each audio segment should be at least 10 to 30 seconds long, because too many closely spaced Sync Marks can confuse the computer.

Step 2: It is suggested that before coding begins, the programmer draft a general plan for the program indicating the relationship between screen (CPU) and audio.

Example: Figure C-2 illustrates how a programmer should create a cassette containing a program that control over an audio track. The example is called FROG:

Step 3: The programmer can start coding the program called FROG, and it will look something like this:

```
10 REM PROGRAM "FROG" TO DEMONSTRATE SYNCHRONIZATION
20 REM OF AUDIO WITH DIGITAL FOR THE CASSETTE SYSTEM
30 REM
40 DIM IN$(20)
50 POKE 54018,52:REM TURN ON MOTOR
60 GRAPHICS 1
70 PRINT #6;"THE PRINCESS AND THE FROG":PRINT #6; .... :REM SET UP THE SCREEN
FOR EVENT 2.
80 GOSUB 1000:REM CHECK SYNC MARK, MAKE SURE THE INTRODUCTION IS SAID.
100 POSITION X,Y:PRINT #6;"YOUR NAME?":REM FOR EVENT 4
105 GOSUB 1000:REM EVENT 5
110 POKE 54018,60:REM STOP MOTOR FOR USER INPUT
120 INPUT IN$:REM WAIT FOR THE USER'S NAME
130 POKE 54018,52
135 PRINT #6,CHR$(125):REM CLEAR THE SCREEN
140 POSITION X,Y:PRINT #6;IN$:PRINT #6; .... :REM DISPLAY SCREEN FOR EVENT 10
150 GOSUB 1000:REM MAKE SURE SPEECH FOR EVENT 10 IS FINISHED
160 PRINT #6; .... :REM READY FOR EVENT 12
```

EVENT	AUDIO	SCREEN	CHECK SYNC MARK	MOTOR MODE
1				
2	'TODAY I AM GOING TO...'	THE PRINCESS & THE FROG GRAPHIC		
3			Yes	
4	'BEFORE I...'	THE PRINCESS & THE FROG GRAPHIC YOUR NAME? XXXX		
5			Yes	
6				STOP
7		WAIT TIL AN INPUT IS RECOGNIZED		
8				
9		CLEAR THE SCREEN		START
10	'NOW LET'S...'	XXXX GRAPHIC		
11			Yes	
12	'ON A CLEAR...'	GRAPHIC		
13				

ROUTINE TO CHECK SYNC MARK: On the tape, non-sync is represented by "MARK" and Sync Mark is represented by "SPACE." (Space is a "0" frequency; it is a lower pitch sound than a Mark which is a "1" frequency. As mentioned before, Mark frequency is 5327 Hz, Space is 3995 Hz). The Check Sync Mark routine continuously watches for a "SPACE" from the serial port. The routine looks like this:

```
1000 IF INT(PEEK(53775)/32+0.5)=INT(PEEK(53775)/32) THEN RETURN: REM CHECK THE 5TH BIT
OF EACH INCOMING BYTE. IF IT IS "0" THEN THE SYNC SPACE IS FOUND.
1010 GOTO 1000
```

ROUTINE TO CONTROL THE MOTOR: The program can turn the cassette motor on and off by poking location 54018 with the data given below:

ON: POKE 54018,52
OFF: POKE 54018,60

Step 4: After the audio script has been roughly written, the programmer should estimate the time

and the tape length required for the designed audio script (including pauses) and program. If the tape length required is too long for one cassette, then either the script or the program will have to be modified to fit into one cassette.

Step 5: Save the program to a master tape, for example "MASTER 1."

Step 6: With the audio script the voice is taped with pauses on another master tape, "MASTER 2."

Step 7: After "MASTER 1" and "MASTER 2" are produced, these two master tapes are merged to produce another master tape called "MASTER 3." "MASTER 3" has the program recorded first, and the audio spliced on the end. Three recording lab machines are needed for this procedure. Make two copies of "MASTER 3."

Step 8: Load the Sync Mark program into the Atari 800 Computer. The purpose of this program is to write continuous Sync Mark ("0" frequency) onto the digital track. The Sync Mark informs the program that an audio segment has been played. Whenever there is a pause indicated on the audio script, a Sync Mark is needed at that place. The finished tape with audio and sync would be as shown in Figure C-3.

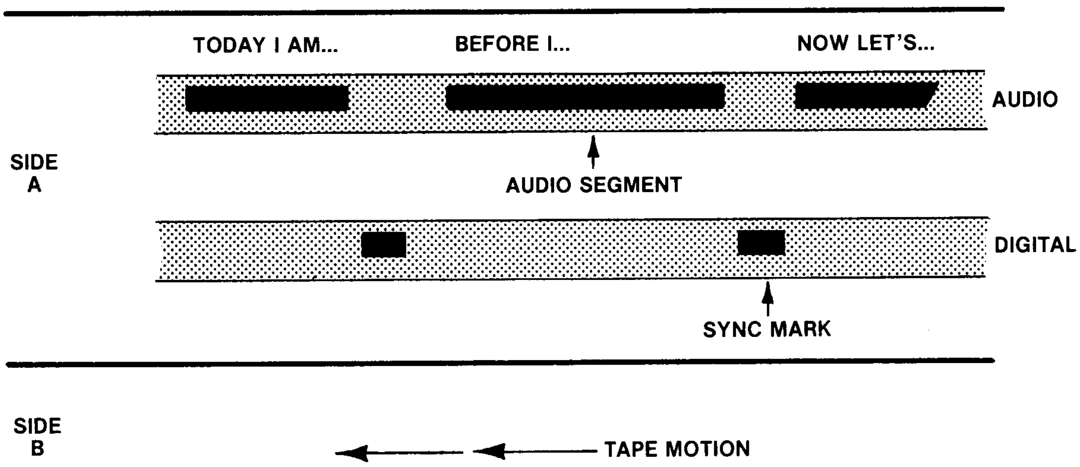


Figure C-3

The Sync Mark program looks like this:

```
10 REM PUSH "START" CONSOL KEY TO
20 REM ADD THE SYNC MARK ONTO THE TAPE
30 REM
40 REM
50 IO=53760 : CONSOLE=53279 : CASS=54018
100 FOR I=0 TO 8
110 READ J : POKE I0+I,J
120 NEXT I
125 REM THE FOR LOOP SETS THE AUDIO FREQUENCY & CHANNEL
130 DATA 5,160,7,160,5,160,7,160,0
140 REM
150 REM I/O IS SETUP; NOW START THE CASSETTE
160 POKE CASS, 52
200 POKE CONSOLE,8
210 IF PEEK(CONSOLE)=7 THEN 230:REM CONSOLE=7 MEANS WRITE MARK,
220 POKE 10+15,11: GOTO 200: REM CONSOLER KEYS NOT PRESSED
230 POKE 10+15,128+11: GOTO 200: REM IF CONSOLE=7 WRITE "SPACE"
```

Step 9: Mount both MASTER 3 tapes in two independent recording machines and rewind both tapes to the splice of program and audio. Configure one recording machine to one ATARI 800 Computer with Sync Mark program loaded. This recording machine is prepared for recording Sync Mark on the digital track. The other recording machine will play back the audio recorded earlier.

Step 10: Type RUN to start the Sync Mark program. At the same time start the recording machines, one for recording, another one for playback. Listen to the audio and press the START key whenever it is indicated by a pause in the audio script.

Step 11: Now the tape is done, with the program recorded followed by the audio and Sync Mark recording. The finished tape is ready for mass production.

DISABLING THE BREAK KEY

We suggest that the programmer disable the BREAK key. This prevents the cassette program from failing when the user accidently hits BREAK The OS will not recover a partial record, unless the user can rewind to the lost record. The disable BREAK key routine looks like this:

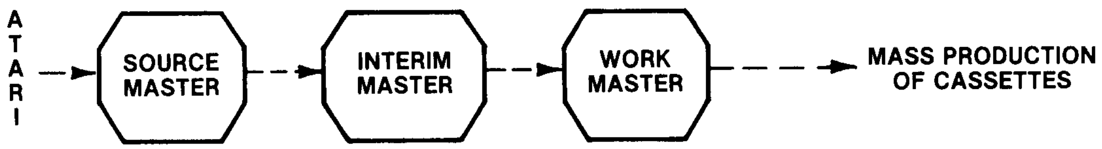
```
4000 X= PEEK(16): IF X 128 THEN 4020
4010 POKE 16,X-128: POKE 53774,X-128
4020 RETURN
```

The disable routine should be called whenever there is a change of graphics mode or any screen open call.

MASS PRODUCTION

The programmer produces one or more MASTER TAPES according to the recording techniques discussed in the forgoing paragraph. All Atari Masters are recorded on open-reel 1/4 track, 1/4-inch tape recorded at 7 1/2 inches per second. The MASTER TAPE is supplied to the duplicator as a SOURCE MASTER.

The duplicator will take the SOURCE MASTER to make a WORK MASTER for the final cassette mass production. The released product will be third generation from the original. The following is a flow of the process:



INTERIM MASTER is recommended for the duplicator, because the WORK MASTER may be destroyed or worn from excessive use. The SOURCE MASTER should be reserved only for emergency need. The INTERIM MASTER is the backup copy for the WORK MASTER.

Mass Production Of Cassettes

At present, ATARI prefers the BIN LOOP method for mass production: The WORK MASTER is copied to produce a LOOP MASTER. The LOOP MASTER may be on 1/4 inch, 1/2 inch, or any tape width. The BIN LOOP is spliced into a CONTINUOUS LOOP with a short clear leader at the splice. It is placed in a high-speed loop master machine which has one or more SLAVE machines. The configuration is shown in Figure C-4

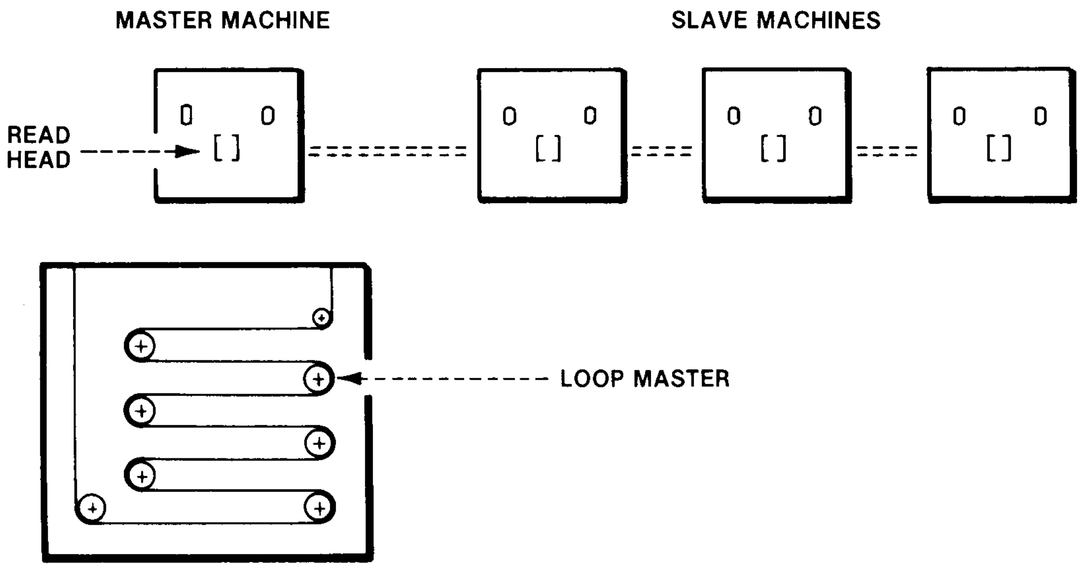


Figure C-4

The LOOP MASTER is repeatedly read. If the duplicator wants to produce 100 cassettes, for example, the length of the tape on the SLAVE MACHINE is measured to the length of the program multiplied by 100. There is a counter on the MASTER machine and it is set to 100.

As the LOOP MASTER is continuously read, the data (all four tracks) is copied onto the SLAVE MACHINE tape.

As the clear section in the LOOP MASTER is sensed, the MASTER machine produces a CUTTING TONE which is recorded on one or more tracks on the SLAVE MACHINE tapes. The counter will then increase by one.

Each finished tape from the SLAVE MACHINE has 100 recorded programs with 100 CUTTING TONES recorded. It is fed into an automatic loading machine which winds the tape into C-Zero cassette shells. The configuration is like this:

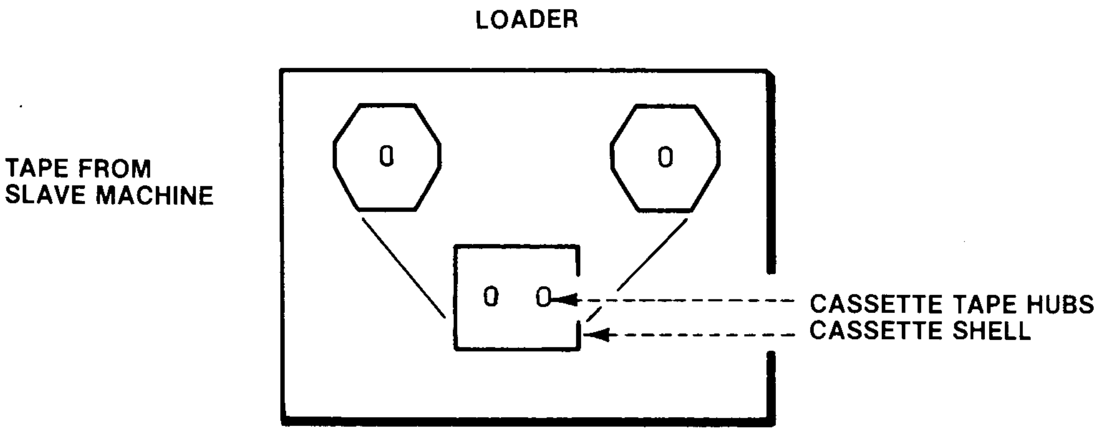


Figure C-5

The cassette shells come with a small loop of leader which is bound to the cassette tape hubs. The loader pulls the leader from the shell, cuts it, and splices the end of the slave machine tape to the leader. The tape hub is used to wind the tape into the shell until the cutting tone is sensed. The slave machine tape is then cut and spliced to the leader on the other hub.

The cassette shell is removed either manually or mechanically from the loader and the tape in the cassette shell is fully wound. The next cassette shell is loaded by the same process.

Quality Control Testing

Any time that a production run is created, samples must be taken from it and verified before it is approved and released.
The QC testing is done normally by taking the first and the last cassette produced. Atari must receive at least 10 samples from each mass production for each master released.

Chapter XIV

Appendix D. Television Artifacts

14 Appendix D. Television Artifacts

This section discusses how to get multiple colors out of a single color graphics mode through the use of television artifacts.

The ANTIC modes with which this can be accomplished are 2,3, and 15. ANTIC mode 2 corresponds to BASIC mode 0, ANTIC mode 15 is BASIC mode 8, and ANTIC mode 3 has no corresponding BASIC mode. Each of these modes has a pixel resolution of one half color clock by one scan line. They are generally considered to have one color and two luminances. With the use of artifacts, pixels of four different colors can be displayed on the screen in each of these modes.

The term TV artifacts refers to a spot or "pixel" on the screen that displays a different color than the one assigned to it.

A simple example of artifacts using the ATARI Computer is shown by entering the following lines:

```
GRAPHICS 8
COLOR 1
POKE 710,0
PLOT 60,60
PLOT 63,60
```

These statements will plot two points on a black background; however each pixel will have a different color.

To understand the cause of these differing colors one must first understand that all the display information for the television display is contained in a modulated television signal.

The two major components of this signal are the luminance, or brightness, and the color, or tint. The luminance information is the primary signal, containing not only the brightness data but also the horizontal and vertical syncs and blanks. The color signal contains the color information and is combined or modulated into the luminance waveform.

The luminance of a pixel on the screen is directly dependent on the amplitude of the luminance signal at that point. The higher the amplitude of the signal, the brighter the pixel.

The color information, however, is a phase shifted signal. A phaseshifted signal is a constantly oscillating waveform that has been delayed by some amount of time relative to a reference signal, and this time delay is translated into the color.

The color signal oscillates at a constant rate of about 3.579 MHz, thus defining the highest horizontal color resolution of a television set. This appears on the screen in the form of 160 visible color cycles across one scan line. (There are actually 228 color cycles including the horizontal blank and sync, and any overscan.)

The term "color clock" refers to one color cycle and is the term generally used throughout the ATARI documentation to describe units of measurement across the screen. The graphics mode 7 is an example of one color clock resolution, where each color clock pixel can be a different color. (There are microprocessor limitations though.)

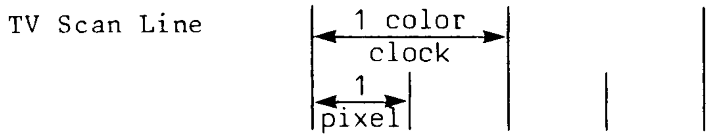
Atari also offers a "high resolution" mode (GRAPHICS 8) that displays 320 pixels across one line. This is generated by varying the amplitude of the luminance signal at about 7.16 MHz, which is twice the color frequency.

Since the two signals are theoretically independent, one should be able to assign a "background" color to be displayed and then merely vary the luminance on a pixel-by-pixel basis. This in fact is the way mode 8 works, the "background" color coming from playfield register 2, and the luminances coming from both playfield registers 1 and 2.

The problem is that in practice the color and luminance signals are not independent. They are part of a modulated signal that must be demodulated to be used. Since the luminance is the primary signal, whenever it changes, it also forces a change in the color phase shift. For one or more color

clocks of constant luminance this is no problem, since the color phase shift will be unchanged in this area. However, if the luminance changes on a half color clock boundary it will force a fast color shift at that point. Moreover, that color cannot be altered from the transmitting end of the signal (the ATARI Computer).

Since the luminance can change on half color clock boundaries, this implies that two false color, or artifact pixel types can be generated. This is basically true. However, these two pixels can be combined to form two types of full color clock pixels. This is illustrated below:



Luminance 0=off 1=on					
	0	1	0	0	1/2 cc pixel color A
	1	0	0	0	1/2 cc pixel color B
	1	1	0	0	1 cc pixel color C
	0	1	1	0	1 cc pixel color D

Figure D-1

Note that each of these pixels requires one color clock of distance and therefore has a horizontal resolution of 160.

The colors A through D are different for each television set, usually because the tint knob settings vary. Thus they cannot be described as absolute colors, for example, red; but they are definitely distinct from each other, and programs have been written that utilize these colors.

TELEVISION ARTIFACTS

To illustrate a simple application of artifacting, refer to the example below. This program draws lines in each of the four artifact colors and then fills in areas using three of the colors. (Note that displaying many pixels of either type C or D next to each other results in the same thing: a line or constant luminance with background color.)

The POKE 87,7 command causes the OS to treat this mode as mode 7 and to use two-bit masks when setting bits in the display memory. To generate color A, use COLOR 1, color B uses COLOR 2, and color C uses COLOR 3. Color D is generated by displaying COLOR 1 to the left of COLOR 2.

```
10 GRAPHICS 8:POKE 87,7:POKE 710,0:POKE 709,14
20 COLOR 1:PLOT 10,5:DRAWTO 10,70
30 PLOT 40,5:DRAWTO 40,70
40 COLOR 2:PLOT 20,5:DRAWTO 20,70
50 PLOT 41,5:DRAWTO 41,70
60 COLOR 3:PLOT 30,5:DRAWTO 30,70
70 FOR X=l TO 3:COLOR X:POKE 765,X
80 PLOT X*25+60,5:DRAWTO X*25+60,70
90 DRAWTO X*25+40,70:POSITION X*25+40,5
100 XIO 18,#6,12,0,"S:l"
110 NEXT X
```

Chapter XV

Appendix E. GTIA

15 Appendix E. GTIA

The GTIA is a new display chip that will someday replace CTIA. Actually it is nothing more than a CTIA with a few more features. It simply provides three additional modes of interpretation of information coming from the ANTIC chip. ANTIC does not require a new mode to talk to GTIA; instead, it uses the high resolution mode $F. GTIA is completely upward compatible with the CTIA. A brief summary of CTIA's features follows so that the differences between CTIA and GTIA can be presented.

The CTIA is designed to display data on the television screen. It displays the playfield, players and missiles, and detects any overlaps or collisions between objects on the screen. CTIA will interpret the data supplied by ANTIC according to six text modes and eight graphics modes. In a static display, it will use the data from ANTIC to display hue and luminance as defined in one of four color registers. The GTIA expands this to use all nine color registers or 16 hues with one luminance or 16 luminances of one hue in a static display.

The three graphics modes of GTIA are simply three new interpretations of ANTIC mode $F, a hi-resolution mode. All three modes affect the playfield only. Players and missiles can still be added to introduce new hues or luminances or to use the same colors and luminances in more than one way. All displays of hues and luminances can still be changed on-the-fly with display list interrupts. The GTIA uses four bits of data from ANTIC for each pixel, called the pixel data. Each pixel is two color clocks wide and one scan line high. Thus, the pixels are roughly four times wider than their height. The display has a resolution of 80 pixels across by 192 down. Each line then requires 320 bits or 40 bytes of memory, the same number of bytes used in ANTIC mode $F. Therefore for a program to run the GTIA modes it must have at least 8K of free RAM for the display.

The GTIA modes are selected by the priority register, PRIOR. PRIOR is shadowed at location $26F hex by the OS and is located at D01B hex in the chip. Bits D6 and D7 are the controlling bits. When neither is set there are no GTIA modes and GTIA operates just like CTIA. When D7 is 0 and D6 is 1, Mode 9 is specified which allows 16 different luminances of the same hue. Remember the pixel data supplied by ANTIC is four bits wide which means 16 different values can be represented. Players and missiles can be used in this mode to introduce additional hues. When D7 is 1 and D6 is 0, Mode 10 is specified. This mode gives nine colors in the display by using the four playfield color registers plus the four player/missile color registers plus the one background color register. When players are used in this mode, the four player/missile color registers are used for them also. When D7 is 1 and D6 is 1, Mode 11 is specified. This mode gives 16 hues with the same luminance again because 16 different values can be represented by four bits. Players and missiles can be used in this mode to introduce different luminances.

PRIOR

D7	D6						

D7	D6	OPTION	
0	0	No GTIA modes (CTIA operation)	(Modes 0-8)
0	1	1 Hue, 16 Luminances	(Mode 9)
1	0	9 Hues/Luminances	(Mode 10)
0	0	16 Hues, 1 Luminance	(Mode 11)

Figure E-1 Bit Pattern in PRIOR selects GTIA

Setting up the new GTIA modes is as simple as setting up the present modes used in CTIA. To

implement the modes from BASIC simply use the commands GRAPHICS 9, GRAPHICS 10, and GRAPHICS 11 for Mode 9, Mode 10, and Mode 11 respectively. In Assembly Language selecting one of these modes is identical to opening the screen for any of the other modes. If you are building your own display list then PRIOR must be set to select the correct mode as in Figure E-1.

Mode 9 produces up to 16 different luminances of the same hue. ANTIC provides the pixel data which selects one of 16 different luminances. The background color register provides the hue. In BASIC this is done using the SETCOLOR command to set the hue value in the upper nybble of the background color register, and to set the luminance value in the lower nybble to all zeroes. The format of the command is

```
SETCOLOR 4,hue value,0
```

where 4 specifies the background color register, "hue value" sets the hue and can be anything from 0 to 15, and 0 will set the luminance part of the register to zero. This has to be done because the pixel data from ANTIC will then be logically OR'ed with the lower nybble of the background color register to set the luminance that appears on the screen. The COLOR command is then used to select luminances for drawing on the screen by using values from 0 to 15 as its parameter. So a BASIC program will include at least the following statements to use Mode 9:

```
            GRAPHICS 9 to specify Mode 9
            SETCOLOR 4,12,0   to initialize the background color register to some hue, in
this case green.
            FOR I=0 TO 15
            COLOR I
            PLOT 4,I+10       some method where the COLOR command is used to vary
luminance.
            NEXT I
```

In Assembly Language use the OS shadow for the background color register $2C8 to set the hue in the upper four bits with hex values from $0 to $F. If CIO calls are used, store the pixel data into the OS register ATACHR located at $2FB. This selects the luminance with hex values from $0 to $F. If you are maintaining your own display data then the pixel data goes directly into the left or right half of the display RAM byte.

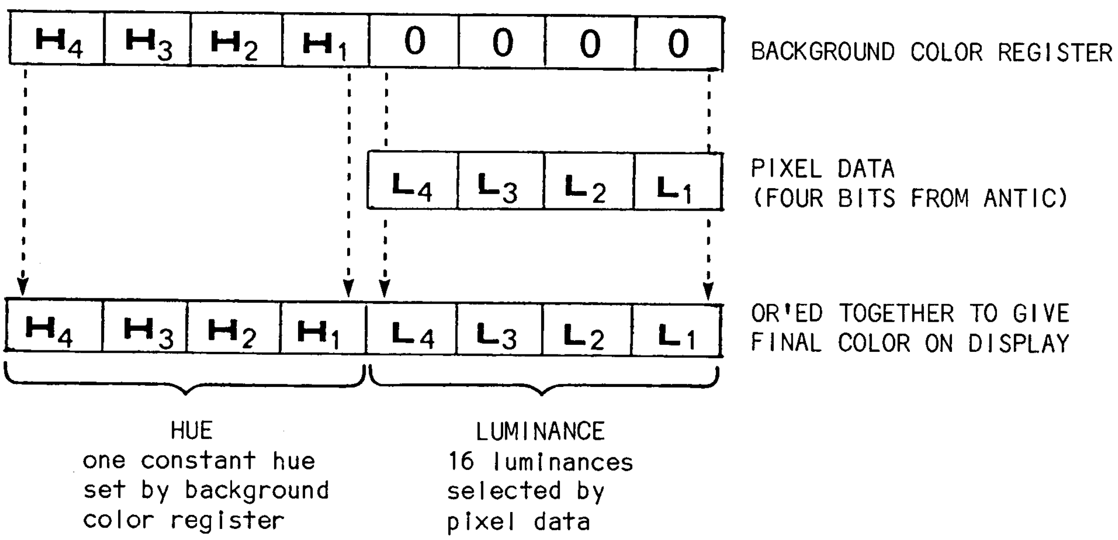


Figure E-2 Background Color Register OR'ed with pixel data to give final color.

Mode 11 is similar to Mode 9 except that it provides 16 different hues all with the same luminance. Again ANTIC will provide the pixel data to select one of 16 different hues. In BASIC the SETCOLOR command is used to set up the single luminance value in the lower nybble of the background color register, and in the upper nybble, the hue value will be set to all zeroes. The format of the command is

```
SETCOLOR 4,0,luminance value
```

where 4 specifies the background color register, 0 sets the upper nybble to zero and "luminance value" sets the value of the luminance and can range from 0 to 15. As with the other graphics modes (except Mode 9), the first bit of the luminance is not used, so effectively only even numbers result in distinct luminances which gives eight different possible luminances in this mode. The COLOR command is used in this mode to select the various hues by using values from 0 to 15 in its parameter. The pixel data from ANTIC will be logically OR'ed with the upper nybble of the background color register to set the hue part of the value that ultimately generates the color on the screen. So a BASIC program using Mode 11 will include at least the following statements:

```
            GRAPHICS 11       to specify Mode 11
            SETCOLOR 4,0,12   to initialize the background color register to some
luminance, in this case very bright
            FOR I=0 TO 15
            COLOR I
            PLOT 4,I+10       some method where the COLOR command is used to vary the hue
            NEXT I
```

In Assembly Language use the OS shadow for the background color register $2C8 to set the luminance in the lower four bits with hex values from $0 to $F. If CIO calls are used, store the pixel data into ATACHR located at $2FB. This selects the hue with hex values from $0 to $F. If you are maintaining your own display data then the pixel data goes directly into the left or right half of the display RAM byte.

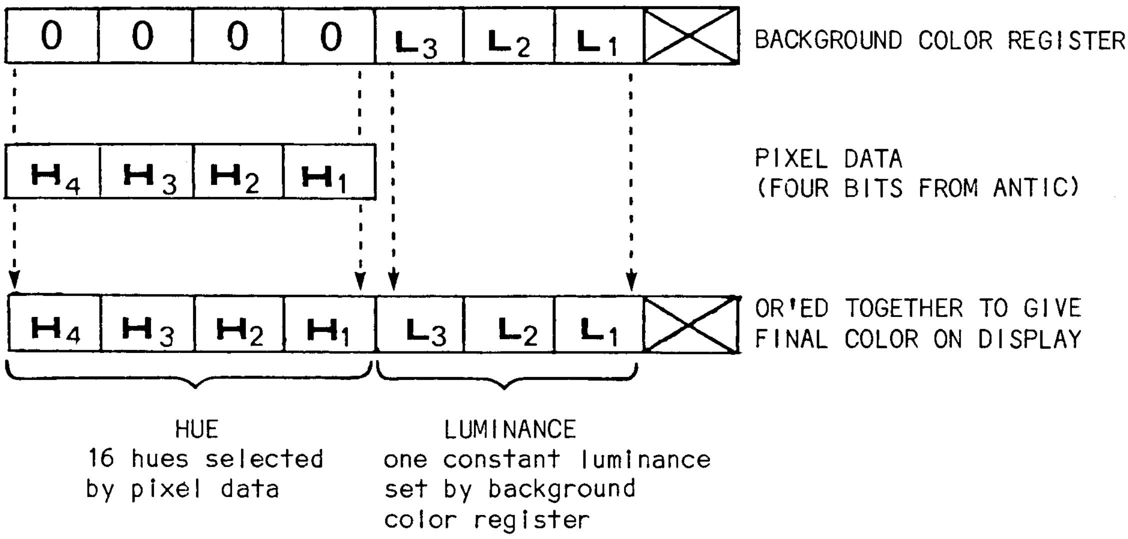


Figure E-3 Background Color Register OR'ed with pixel data to give final color.

Mode 10 will allow all nine color registers to be used in the playfield at one time. Each color register to be used must be set to some combination of hue and luminance. The pixel data from ANTIC is used in this mode to select one of the color registers for display. In BASIC the SETCOLOR command can be used as described in the BASIC Reference Manual to set the colors in the background and the four playfield registers. These can also be set by using the POKE instruction to addresses 708-712 where the four playfield registers and the background register are located. The POKE instruction must be used to set the four player/missile color registers at locations 704-707. The COLOR command is used to select the color register desired. The only meaningful values for its argument are 0 to 8. A problem arises with this mode. ANTIC supplies four bits of data per pixel, as it does with Modes 9 and 11. This allows for the selection of 16 color registers. However, only nine color registers exist in the hardware. An illegal data value between 9 and 15 will select one of the lower value color registers. A BASIC program using mode 10 will include:

1. a GRAPHICS 10 command to specify Mode 10;

2. a set of POKE instructions to put hues and luminances into the color registers, OR a combination of SETCOLOR commands and POKE instructions to do that;

3. a COLOR command to select the desired color register.

In Assembly Language, store the pixel data in ATACHR ($2FB) or directly into the display RAM byte as in Modes 9 and 11. In this mode the pixel data can range from 0 to 8 and selects one of the nine color registers.

COLOR STATEMENT VALUE	**COLOR REGISTER USED**	**OS SHADOW**
0	D012	2CO
1	D013	2C1
2	D014	2C2
3	D015	2C3
4	D016	2C4
5	D017	2C5
6	D018	2C6
7	D019	2C7
8	D01A	2C8

Figure E-4 Color Register numbers and locations and COLOR command reference.

An important question arises in conjunction with GTIA concerning compatibility. GTIA is fully upward compatible with the CTIA and all software that runs on a CTIA system will run the same way on a system with GTIA. This means you still have the full use of players and missiles, still have collision and overlap detection and display list interrupts. The GTIA graphics modes are fully supported by the OS and all graphics commands and utilities that run in the CTIA modes can be used in GTIA modes.

More colors are available to display at one time on the screen. Sixteen color changes can occur on one line totally independent of processor intervention. This is actually better than what could be done with display list interrupts which could give at most only 12 color changes per line. Much finer contour and depth can be represented using the shading available in Mode 9. This means three dimensional graphics can be realistically displayed.

On the other hand, there are some disadvantages. GTIA modes are map modes, there can be no text displayed in these modes. A custom display list must be used to switch to a mode that supports character displays. The GTIA pixel is a long, skinny horizontal rectangle (4:1, width to height) and does not represent curved lines well. Because each pixel uses four bits of information, GTIA requires nearly 8K of free RAM to operate. Although it is upward compatible, it is NOT downward compatible. Thus programs which use GTIA modes will NOT produce correct displays on computers that have CTIA'S. They may well be recognizable but will not be as colorful. There is no way currently for a program to determine whether or not a GTIA is present in a system. Finally, color artifacts produced by a GTIA system will not be identical to the color artifacts produced on the same television with a CTIA system.

Chapter XVI

Glossary

16 Glossary

$
This symbol in front of a number indicates that the number should be interpreted as hexadecimal.

ANTIC
This is a separate microprocessor, contained within the ATARI 400/800 Computers, which is dedicated to the television display. ANTIC is user- programmable with an instruction set, a program (the "display list"), and data (the "display memory").

ATTRACT MODE
This is a feature provided by the operating system which, after nine minutes without a key being pressed, cycles the colors on the screen through random hues at lowered luminances. This ensures that a computer left unattended for several hours doesn't burn a static image into the television screen.

BACKGROUND
The area of the television screen display upon which player-missile graphics objects or playfield objects and/or text are projected. Background has its own user-definable color.

BCD
Acronym for Binary Coded Decimal. A numbering system in which each number is broken into a sequence of decimal digits. These decimal digits are then coded into binary, a task which requires four bits per digit, and stored in the resultant form. In the ATARI Computer, two such digits are stored in each 8-bit byte.

BORDER
In BASIC Mode 0, this is the area of the television screen display which is formed by the four edges of the screen. The border takes background color.

BRKKEY
A flag set when the OS senses that the BREAK key is typed. BRKKEY'S normal value is $FF -- if it changes, then the BREAK key has been pressed.

BYTE COUNT
This is the file pointer's position within a sector on diskette.

CASSETTE BOOT FILE
A standard or user-created file which boots from cassette at power-up or SYSTEM RESET.

CHARACTER GRAPHICS
The technique of redefining the individual characters of a character set to form graphics images instead of text characters.

CHARACTER IMAGE
The unique 8 X 8 pixel grid which defines a particular character's shape.

CHARACTER MODE
This is a specific type of ANTIC display mode which displays screen display memory data bytes as characters, using a character set. There are six ANTIC character modes, three of which are accessible from BASIC.

CHARACTER NAME BYTE
A one-byte ANTIC display memory value which selects a unique character within the current character set using the character's sequential position in that set.

CHARACTER SET INDIRECTION
The technique of specifying to ANTIC a particular character set to be used by placing that set's beginning page address into CHBAS.

CHBAS

The OS shadow location which ANTIC uses to find the current character set which is to be used for character display modes. CHBAS is at decimal address 756.

CHECKSUM
This is a single byte sum of all the bytes in a record (either disk I/O or cassette I/O). For cassette I/O, this includes addition of the two marker characters, computed with end-around carry.

CIO
Acronym for Central I/O system routine. CIO routes I/O control data to the correct device handler and then passes control to the handler. CIO is also the common entry point for most of the OS I/O functions.

COARSE SCROLLING
The process of altering the display list LMS (Load Memory Scan) address bytes in order to vertically or horizontally scroll the screen image, one byte at a time. This is accomplished by adding 1 to or subtracting 1 from the LMS address bytes.

COLDSTART
Synonym for the power-up process which performs a series of system database initializations when the computer power switch is turned on. After coldstart, the system surrenders control to the user.

COLLISION
This occurs when a player or missile image coincides with another image. There are 60 possible collisions and each one has a bit assigned to it that can be checked. These bits are mapped into 16 registers in CTIA (with only the lower 4 bits used).

COLOR
One of 128 values obtained from a hue-luminance combination which is stored in a color register.

COLOR CLOCK
The standard unit of horizontal distance on the television screen. There are 228 color clocks in a horizontal scan line, but only 160 are displayed in a normal width playfield.

COLOR REGISTER
A hardware register (with corresponding OS shadow location) used to define the color for various portions of the screen display. There are nine color registers available on the ATARI Home Computer.

COLOR REGISTER INDIRECTION
The technique of specifying a particular color by pointing to its color register rather than directly specifying it.

COLOR SIGNAL
This contains the color information which is combined with the primary signal to form the modulating television signal. The color signal oscillates at 3.579 Mhz.

COLRSH
A zero-page location ($4F) set up and updated by the OS during vertical blank interrupts for ATTRACT mode processing. When ATTRACT mode is in force, COLRSH is given a new random value every 4 seconds.

COMMAND
In BASIC, this is the first executable token of a BASIC statement that tells BASIC to interpret the tokens that follow in a particular way.

CONSTANT
In BASIC, this is a 6-byte BCD value preceded by a special token. This value remains unchanged throughout the program execution.

CONTROL BYTE

In cassette I/O, this is part of every record. It contains one of three possible values.

CTIA
A television interface chip which is controlled primarily by ANTIC. CTIA converts ANTIC's digital commands into a signal that is sent to the television.

CURRENT STATEMENT
In BASIC, this is the current token within a line of the Statement Table.

CYCLE STEALING
This occurs when ANTIC halts 6502 processing in order to perform DMA functions for memory refresh and screen display purposes.

CYCLIC ANIMATION
The technique of repetitively flipping through colors, graphics images, or character graphics sets to animate screen images.

DCB
Acronym for Device Control Block. The DCB is used by the I/O subsystem to communicate between the device handier and SIO.

DEVICE HANDLERS
Routines present in OS ROM which are called through CIO (as long as the handler has an entry in HATABS) to communicate with particular devices. Currently supported are the display editor, the screen, the keyboard, the printer, and the cassette. More handlers can automatically boot in during power-up.

DEVICE SPEC
A special HATABS code which specifies a particular I/O device.

DIAGONAL SCROLLING
This results from the combination of horizontal and vertical scrolling of the screen image.

DISPLAY LIST
ANTIC's "program" defined by the user or provided automatically (through a GRAPHICS command) by BASIC. The display list specifies where the screen data may be found, what display modes to use to interpret screen data, and what special display options (if any) should be implemented.

DISPLAY LIST INTERRUPT
A special ANTIC display list instruction which interrupts the 6502 microprocessor during the drawing of the screen image, allowing the 6502 to change the screen parameters.

DISPLAY MODE
Either a BASIC or ANTIC methodology for interpreting text or map data bytes in screen memory and displaying them on the screen. ANTIC provides 15 display modes; BASIC, through the OS, supports only 9 of these modes.

DLI VECTOR
This is a 2-byte vector (low byte, high byte) to the Display List Interrupt service routine. This vector is set by the user and is located at [512,513] decimal.

DMA
Direct Memory Access. This occurs when ANTIC halts the 6502 and takes control of the system buses to fetch an instruction or data byte from memory.

DMACTL
The hardware register whose bit settings control the use of DMA by the ANTIC chip. This affects, among other things, player vertical resolution and player-missile graphics enabling.

DOS

Acronym for Disk Operating System which is an extension of the OS that allows the user to access disk drive mass storage as files.

DOUBLE-LINE RESOLUTION
A unit of vertical resolution for a player in player-missile graphics. Each player byte occupies two horizontal scan lines on the screen, and each player table is 128 bytes long.

DRKMSK
A zero-page ($4E) location set up and updated by the OS during vertical blank interrupts for ATTRACT mode process color register's value. This ensures a low luminance for ATTRACT mode.

DUP
Acronym for Disk Utility Package. DUP is a set of utilities for disk drive usage, familiarly seen as the DOS menu. DUP executes commands by calling FMS through CIO.

DYNAMIC DISPLAY LIST
This is an ANTIC display list which the 6502 changes during vertical blank periods, allowing for even greater flexibility in the screen display.

EOL
In BASIC, "End-of-Line", a character with the value $9B.

FILE
In cassette I/O, this consists of a 20-second leader of the mark tone plus any number of data bytes, and end-of-file. In diskette I/O, this consists of a number of sectors linked by pointers (125 data bytes per sector).

FILE POINTER
For diskette I/O, this is a value which indicates the current position in a file by specifying the Sector Number and the Byte Count. DOS keeps a file pointer for every file currently open.

FINE SCROLLING
The process of horizontally or vertically scrolling a screen image in color clock or scan line increments. The horizontal scrolling and vertical scrolling hardware registers must be used to fine scroll.

FMS
File Manager System. FMS is a nonresident device handler which supports some special CIO functions.

FONT
A collection of characters which constitutes a character set. These characters can be either text or graphics images.

FOREGROUND
Equivalent to playfield, the area of the screen which directly overlays the background of the screen. Foreground is formed by map displays and/or text.

FORMAT
A resident disk handler command that clears all the tracks on diskette.

FUNCTION
In BASIC, a token that when executed returns a value to the program.

GRAPHICS INDIRECTION
A special feature of the ATARI Computer which allows color register and character set generality by using indirect pointers to color and character set values.

HATABS
The device handler entry point table which is used by CIO. HATABS is located at $031A.

HORIZONTAL BLANK
This is the period during which the electron beam (as it draws the screen image) turns off and returns from the right edge of the screen to the left edge.

HORIZONTAL POSITION REGISTER
A special register which contains a user-definable value for the horizontal position of a player in player-missile graphics. This value is measured in units of color clocks.

HORIZONTAL SCAN LINE
The fundamental unit of measurement of vertical distance on the screen. The scan line is formed by a single trace of the electron beam across the screen.

HORIZONTAL SCROLL ENABLE BIT
This is bit D6 of the ANTIC display instruction which enables horizontal scrolling through the HSCROL register.

HORIZONTAL SCROLLING
This is the process of sliding the screen window to the left or right over display memory in order to display more information than could be seen with a static screen. Both coarse and fine horizontal scrolling and available.

HSCROL
This is the horizontal fine scrolling register located at $D404, containing the number of color clocks by which a line is to be horizontally scrolled.

HUE
The upper nybble value of a color register's color. There are 16 possible hues ($0 to $F) which in combination with a luminance value constitute distinct colors. Examples of hues are black, red, and gold.

IMMEDIATE MODE
In BASIC, the mode where the input line is not preceded by a line number. BASIC immediately executes the line.

INPUT BAUD RATE
For cassette I/O, this is assumed to be a nominal 600 baud (physical bits per second). However, this rate is adjusted by SIO to account for drive motor variations, stretched tape, etc.

INPUT LINE BUFFER
In BASIC, from $580 to $5FF.

INTER-RECORD GAP
For cassette I/O records, this consists of the Post-record Gap of a given record followed by the Pre-record Write Tone of the next record.

I/O
Input/Output.

IOCB
Acronym for Input/Output Control Block. There are eight of these whose function is to communicate between the user program and CIO.

IRQ
Maskable (can be enabled or disabled by the 6502) interrupts such as the Break Key IRQ.

IRQEN
The write-only register that contains the IRQ enable/disable bits. IRQEN is shadowed at POKMSK.

KERNEL
A primitive software/hardware technique which consists of a 6502 program loop which is precisely

timed to the display cycle of the television set. The kernel code monitors the VCOUNT register and consults a table of screen changes catalogued as a function of VCOUNT values so that the 6502 can arbitrarily control all graphics values for the entire screen.

LINE
In BASIC, a line consists of one or more BASIC statements preceded either by a line number in the range of 0 to 32767, or an immediate mode line with no line number.

LOMEM
In BASIC, this is the pointer ([80,81] decimal) to a buffer used to tokenize one line of code. The buffer is 256 bytes long, residing at the end of the operating system's allocated RAM.

LSI
Acronym for Large Scale Integration. This refers to a technology for manufacturing silicon chips. LSI chips are the largest and most powerful chips in mass production; they contain many thousands of components.

LUMINANCE
The lower nybble of a color register's color. There are eight even- numbered values for luminance ($0 to $F, even values only) which in combination with hue values produce the 128 colors available on the ATARI 400/800 Computer.

MAP MODE
This is a specific type of ANTIC display mode using simple colored screen pixels instead of characters for the screen display. There are eight ANTIC map modes, with varying degrees of resolution. Six of these are callable from BASIC.

MARK
For cassette I/O, this is a 5327-Hz frequency.

MARKER CHARACTER
For cassette I/O, this is a 55 (hex) value whose purpose is for adjusting the baud rate. Including the start and stop bits, each marker character is 10 bits long.

MEMTOP
In BASIC, a pointer ([90,91] decimal) to the top of application RAM, the end of the user program. Program expansion can occur from this point to the end of free RAM, which is defined by the start of the display list. This MEMTOP is not the same as the OS variable called MEMTOP.

MISSILE
A one-dimensional image in RAM used in player-missile graphics which is 2 bits wide. There is a maximum of four missiles, one for each player.

MODE LINE
A collection of horizontal scan lines for screen displays. Depending upon the BASIC or ANTIC display mode in effect, a mode line will be composed of varying numbers of scan lines. By the same token, depending upon the display mode, a screen image will be composed of varying numbers of mode lines.

MONITOR
A program in ROM that handles both the system power-up and SYSTEM RESET sequences.

NARROW PLAYFIELD
A screen display width option equal to a width of 128 color clocks.

NMI
Non-Maskable Interrupt (i.e., cannot be disabled by the 6502). The Display List Interrupt and the Vertical Blank Interrupt are both NMIs. These can be disabled with the ANTIC NMIEN register.

NMIEN
The Non-Maskable Interrupt Enable register which controls enabling of various NMI interrupts such

as the Display List Interrupt (DLI).

NORMAL IRG MODE
In cassette I/O, this is a mode where the tape always comes to a stop after each record is read. If the computer stops the tape and gets its processing done fast enough, then the next read may occur so quickly that the cassette deck may see only a slight dip in the control line.

NORMAL PLAYFIELD
A screen display width option equal to a width of 160 color clocks.

OPERATOR
In BASIC, any one of the 46 tokens that in some way move or modify the values that follow them.

OPERATOR STACK
In BASIC, a software stack where operators are placed when an arithmetic BASIC expression is being evaluated.

OVERSCAN
The "spreading out" of a television image by the raster scan method of display so that the edges of the picture are off the edge of the television tube. This guarantees no unsightly borders in the television picture.

PIA
Acronym for Peripheral Interface Adaptor. This is an LSI chip which interfaces the 6502 with external devices. The joystick pins of the four user ports are connected to a PIA inside the computer.

PIXEL
The smallest screen graphics unit addressable in a particular display mode. It is a square whose size depends on the display mode.

PLAYER
A one-dimensional RAM image used in player-missile graphics which can be 128 bytes (double-line resolution) or 256 bytes (single-line resolution) long. The player appears as a vertical band 8 pixels wide stretching from the top of the screen to the bottom. There is a maximum of four independent players.

PLAYER COLOR
The color of a player in player-missile graphics. Each of the four independent players has its own color stored in its associated color register.

PLAYER-MISSILE AREA
A RAM area that contains the images of the four players and four missiles of player- missile graphics, as well as some extra RAM. The player-missile area must be on a 1K boundary for single-line resolution players or a 2K boundary for double-line resolution players.

PLAYER-MISSILE GRAPHICS
Atari's solution for simplifying animation by creating an image (a player or missile) which is one-dimensional in RAM but two- dimensional on the screen.

PLAYFIELD
The area of the screen which directly overlays the background of the screen. Map graphics and/or text form this playfield.

PLAYFIELD ANIMATION
The technique of animating an object by moving its image bytes to new locations in screen memory, and then erasing the bytes of the old image before displaying the new image.

PMBAS
A register that points to the beginning of the player-missile area.

POKEY
A digital I/O chip that handles the serial I/O bus, audio generation, keyboard scan-, and random number generation. POKEY also digitizes the resistive paddle inputs and controls maskable Interrupt (IRQ) requests.

POKEY TIMERS
These are hardware timers within POKEY. Unlike System Timers, which are maintained by the OS software and are fixed, the POKEY chip timers are clocked by frequencies set by the user.

POST-RECORD GAP
A pure mark tone frequency used as a post-record delimiter in cassette I/O.

PRE-RECORD WRITE TONE
A pure mark tone frequency used as a pre-record delimiter in cassette I/O.

PRIMARY SIGNAL
This contains the luminance information -- brightness data, horizontal and vertical syncs and blanks -- of the modulated television signal.

PRIORITY-CONTROL REGISTER
Also known as PRIOR, and shadowed at GPRIOR. This register specifies which playfield, player, or background images have priority in the case of image overlaps during the screen display process.

RAM VECTOR
Alterable system vector that contains 2-byte addresses to system routines, handler entry pointers, or to initialization routines. RAM vectors are initialized at power-up and SYSTEM RESET.

RASTER SCAN
A television display system that uses an electron beam generated at the rear of the television tube. The beam sweeps across the screen in a regular left-to-right, top-to-bottom fashion.

RECORD
For diskette I/O, a group of bytes delimited by EOLs ($9B). For cassette I/O, this is a group of 132 bytes which is composed of two marker characters for cassette speed measurement, a control byte, 128 data bytes, and the checksum byte.

RESIDENT DISK HANDLER
The fundamental software in the OS ROM containing the absolutely essential disk handler routines. This software performs five important low- level disk I/O functions such as FORMAT, READ SECTOR, WRITE SECTOR, WRITE/VERIFY SECTOR, and STATUS.

ROM VECTOR
Unalterable system vector that contains JMP instructions to system routines. The ROM vector allows a programmer to write software that uses the OS routines without running the risk of the routines being made unworkable by new releases of the OS ROM.

RTCLOK
One of the system timers which is 3 bytes in length and is updated during immediate VBLANK. RTCLOK can be used as a reference clock for an application program.

RUNSTK
In BASIC, a pointer ([8E,8F] decimal) to the Run Time Stack.

RUN TIME STACK
In BASIC, a software stack that contains GOSUB and FOR/NEXT return address entries.

SCREEN MEMORY
A RAM area used by the 6502 to store bytes of data that will be fetched (by DMA) by ANTIC to be interpreted and eventually displayed as images on the screen.

SECTOR
On a diskette, this is a 128-byte physical area. The diskette contains 40 tracks with. 18 sectors per track.

SECTOR NUMBER
A value from 1 to 719 that specifies the sector to which the file pointer is currently pointing.

SETVBV
A system routine that sets the system timers and sets user-definable interrupt vector addresses without danger of crashes due to interrupts in mid-process.

SHADOWING
A process in which values are moved between hardware locations and RAM locations, thereby allowing the program to monitor the contents of write-only hardware registers or check the inputs form read-only hardware registers.

SHORT IRG MODE
In cassette I/O, this means the tape is not stopped between records. The BASIC commands "CSAVE" and "CLOAD" both specify this mode.

SINGLE-LINE RESOLUTION
A unit of vertical resolution for a player in player-missile graphics. Each player byte occupies one horizontal scan line on the screen, and each player table is 256 bytes long.

SIO
Serial I/O system routine which handles communication between the serial device handlers in the computer and devices on the serial bus (cassette, printer, disk drive, and RS-232).

SIO INTERRUPTS
These are three IRQ interrupts used by SIO to send and receive serial bus communications to serial bus devices. These three are VSERIR (Serial Input Ready), VSEROR (Serial Output Needed), and VSEROC (Transmission Finished).

SOUND REGISTER
Audio-producing hardware in the ATARI Home Computer System which contains frequency, volume, and distortion information, but not duration.

SPACE
For cassette I/O, this is a 3995-Hz frequency output to the cassette tape as a delimiter in conjunction with mark tones.

STARP
In BASIC, the pointer ([8C,8D] decimal) to the String Array Area.

STATEMENT
In BASIC, this is a complete "sentence" of tokens that causes BASIC to perform some meaningful task. In LIST form, statements are separated by colons.

STATEMENT TABLE
In BASIC, this is a block of data that includes all the lines of code that have been entered by the user and tokenized by BASIC. This table also includes the immediate mode line.

STMCUR
In BASIC, the pointer ([8A,8B] decimal) to the current BASIC statement.

STMTAB
In BASIC, this is the pointer ([88,89] decimal) to the Statement Table.

STRING ARRAY AREA
In BASIC, this block contains all the string and array data.

SYNC MARK
This is a 3995-Hz space frequency used as a sort of "end-of-record" marker for audio tracks on the cassette. In applications software it is useful for synchronizing the computer screen display with cassette audio.

SYSTEM. DATABASE
This is an area that occupies RAM pages 0 through 4, containing many locations that store information of importance to the operating system.

SYSTEM TIMER
A timer provided by the ATARI 400/800 Computers that runs at the frequency of the television frame which for North American televisions (NTSC) is 59.923334 Hz. European (PAL) televisions run at 50 Hz. There are six system timers, and they are clocked as part of the vertical blank process.

TELEVISION ARTIFACT
A pixel on an NTSC screen, one color clock wide, that contains color not assigned by the computer. This color is derived from internal oddities of color television displays. Artifacting is possible in ANTIC modes 2,3, and 15 which correspond to BASIC modes 0, no mode, and 8.

TEXT WINDOW
On a screen display, this is a two-dimensional area set aside for character displays.

TOKEN
In BASIC, an 8-bit byte containing a particular execution code.

TOKENIZING
In BASIC, this is the process of getting a line of ATASCII character input and creating a series of 8-bit bytes which contain tokens, meaningful execution codes.

VARIABLE
In BASIC, a token that is an indirect pointer to an entries in variable tables that contain the variable name and the variable value.

VARIABLE NAME TABLE
In BASIC, this is the table containing a list of all the variable names that have been entered in a program.

VARIABLE VALUE TABLE
In BASIC, this table contains the numerical value of each variable.

VBREAK
This is the 6502 BRK instruction IRQ vector. Whenever a $00 opcode (the software break instruction) is executed, this interrupt occurs. VBREAK normally points to an RTI instruction.

VCOUNT REGISTER
The ANTIC register which keeps track of which horizontal scan line ANTIC is displaying.

VDSLST
This is the Display List Interrupt NMI vector located at [$0200,$0201].

VERTICAL BLANK
The period during which the electron beam (as it draws the screen image) returns from the bottom of the screen to the top. This period is about 1400 microseconds in duration.

VERTICAL BLANK INTERRUPT
A non-maskable interrupt which occurs every 60th of a second during the vertical blank time of the television display. In responding to this interrupt, the OS performs various housekeeping functions such as shadowing color registers.

VERTICAL SCROLL ENABLE BIT

This is bit D5 of the ANTIC display list instruction byte which enables vertical fine scrolling through VSCROL ($D405), the vertical fine scroll register.

VERTICAL SCROLLING
The process of vertically "rolling" the display screen "window" over a larger amount of screen data in display memory than can be displayed by a static screen window. Both coarse and fine vertical scrolling are available on the ATARI 400/800 Computers.

VIMIRQ
This is the immediate IRQ vector. All IRQs vector through this location. VIMIRQ normally points to the IRQ handier. This vector can be "stolen" to do user IRQ processing.

VINTER
This is the Peripheral Interrupt IRQ vector. The interrupt line is also available on the serial bus. VINTER normally points to an RTI instruction.

VKEYBD
This is the keyboard IRQ vector which is activated by pressing any key except BREAK. This vector normally points to the OS's own keyboard IRQ routine.

VNTD
In BASIC, this is the pointer ([84,85] decimal) to the Variable Name Table Dummy end. BASIC uses this pointer to indicate the end of the name table. This pointer normally points to a dummy zero byte when there are less than 128 variables. When 128 variables are present, this points to the last byte of the last variable name.

VNTP
In BASIC, the pointer ([82,83] decimal) to the Variable Name Table.

VPRCED
This is the Peripheral Proceed IRQ vector. The proceed line is available to peripherals on the serial bus. This IRQ is unused at the present and normally points to an RTI instruction.

VSCROL
This is the vertical fine scroll register located at $D405. Into VSCROL the user stuffs the number of scan lines by which the screen line is to be vertically scrolled.

VSERIN
This is the POKEY serial Input Ready IRQ vector.

VSEROR
This is the POKEY serial Output Ready IRQ vector.

VTIMR1
This is the POKEY timer 1 IRQ vector.

VTIMR2
This is the POKEY timer 2 IRQ vector.

VTIMR4
This is the POKEY timer 4 IRQ vector.

VVBLKD
This is the Vertical Blank Deferred NMI interrupt vector located at [$0224,$0225].

VVBLKI
This is the Vertical Blank Immediate NMI interrupt vector located at [$0222,$0223].

VVTP
In BASIC, this is the pointer ([86,87] decimal) to the Variable Value Table.

WARMSTART
Another name for SYSTEM RESET routine. The warmstart initializes most of the system vectors but does not check RAM size.

WIDE PLAYFIELD
A screen display width option equal to a width of 192 color clocks.

WSYNC
Wait for Horizontal Sync of the electron beam which is drawing the screen image. The WSYNC register, when written to in any way, pulls down the RDY line on the 6502 microprocessor, freezing the 6502 until the electron beam drawing the screen image returns to the left edge of the screen.

ZERO-PAGE
In the ATARI Home Computer System, this is the stretch of memory which spans locations $0000 to $00FF.

ZIOCB
Zero-page I/O Control Block is used to communicate I/O control data between CIO and the device handlers.